Space Force Duo

[Type the document subtitle]

Javier A. Garcia
Charlie De Vivero
12/10/2009

Space Force Duo is a one- or two-player video game, similar in style to the classic arcade games Galaxian (1979), and Galaga (1981). It allows for two players to play a cooperative game, flying space ships and shooting down enemies. Users log in to the game, and are identified by an RFID tag system. The game is played on an FPGA platform, and connects to a second FPGA platform through a RS-232 Serial Port. In a two-player game, a host terminal and a client terminal is determined. The game features up to three enemy ships on screen at a time, and the two-player game allows for two-way voice communication. Two different kinds of AI-driven enemy ships assault the player, and a kill score is kept track of. The player(s) win the game after reaching a certain score. Our final implementation of Space Force Duo demonstrates exceptional data communication over the serial port, though not error-free. The game is indeed playable, though not without its minor glitches and bugs. The RFID system was not fully implemented, and instead, users are identified by manually setting switches to a corresponding binary code.

Table of Contents

1 Overview ... 4
1.1 Objectives ... 4
1.2 Specifications ... 4
1.3 Design Overview ... 5
2 Description ... 5
2.1 Master Finite State Machine (Javier) ... 6
2.2 Game Logic (Javier) ... 8
2.2.1 Player Logic ... 9
2.2.2 Enemy Logic ... 9
2.2.3 Packets Logic ... 10
2.2.4 Collision Detection Logic ... 10
2.3 Video Controller (Javier) ... 11
2.3.1 Pixel Generator Module ... 11
2.3.2 Ship Sprites Modules ... 11
2.3.3 Bullet Sprites Modules ... 13
2.3.4 Character String Display Module ... 13
2.4 Audio Controller (Charlie) ... 14
2.4.1 Audio Recorder ... 14
2.4.2 Record Buffer ... 15
2.4.3 Play Buffer ... 15
2.4.4 Audio Player ... 15
2.5 Network Controller (Charlie) ... 15
2.5.1 Network Transmitter ... 16
2.5.2 Serializer ... 18
2.5.3 De-Serializer ... 19
2.5.4 Network Receiver ... 19
2.6 RFID Interface (Charlie) ... 19
3 Conclusion ... 20
3.1 Testing and Debugging ... 20
3.2 Results ... 20
4 References ... 21
Appendix: Verilog Code ... 22
A. Top Level Module ... 22
B. Audio Controller ... 33

Audio Recorder 33
Record Buffer 33
Play Buffer 34
Audio Player 35
C. Network Controller 35
Network Transmitter 35
Serializer 41
De-Serializer 42
Network Receiver 43
D. Master FSM 47
E. Game Logic 49
F. Video Controller 69
Pixel Generator 69
Ship Sprite Module 82
Ship Sprite Module 83

List of Figures

Figure 1: Top-Level Block Diagram 5
Figure 2: Master FSM Diagram 7
Figure 3: Player 1 (left) and Player 2 (right)........ 12
Figure 4: Enemy 1 (Top), Enemy 2 (Middle), and Enemy 3 (Bottom)........ 12
Figure 5: Audio Controller Block Diagram 14
Figure 6: Network Controller Block Diagram........ 16
Figure 7: Packet Structures........ 17

1 Overview

1.1 Objectives

The *Space Force Duo* game system was designed to meet the following objectives:

- Implement a "space-ship shooter game" where the player takes control of a space ship, assaulted by waves of incoming enemy ships.
- Enable two-player cooperative play through two separate FPGA platforms, communicating through serial port.
- Allow for voice communication between the two players.
- Require players to log in to the game using an RFID tag system.

1.2 Specifications

Our game system has certain high-level specifications that we must achieve with our modular designs.

- The game runs at a screen resolution of 1024 by 768 pixels (XVGA format), at a refresh rate of 60 Hz.
- To simplify the clock counts needed for the display-rendering signals, our whole system runs on a 64.8 MHz clock.
- A maximum of 10 sprites can be rendered on screen at any given point in time, excluding text characters. These 10 sprites will consist of 5 space ships, and 5 bullets.
- For a one-player game, up to 4 enemy ships will be rendered, along with the player ship. For a two-player game, up to 3 enemy ships can be rendered, along with 2 player ships.
- Each ship can at most fire one bullet at a time.
- A two-player game consists of two labkits running the same code. When the host is chosen, the other labkit becomes the client.
- Host processes all game data, and the client receives data to be displayed on screen.
- Voice communication will consist of recording audio at 48 kHz and downsampling to 6 kHz. Anti-aliasing filtering will be applied, along with low-pass reconstruction for upsampled playback.
- Network system will packetize data and send over a RS-232 serial port.
- Serial transmission will run at 225kbps.
- Audio packets will consist of eight 8-bit samples (8 bytes of data).
- Game packets are of different kinds: Sprite state data (16 bytes of up to 5 sprite screen coordinate positions), game state data (1 byte describing current state of game), client input data (1 byte describing button inputs on client side), ID tag data (10 bytes uniquely identifying the user that is logged in the game).

1.3 Design Overview

Though our game implements a host-client system, and consequently the two labkits will have different roles in processing the game, the same exact Verilog code will run on the two machines. There will be “waiting” screens whereby a player can log in to the game system. Whichever user logs in to the game first determines the machine as the host. Once a host is determined, the host communicates to the other labkit to accept the role of client.

All game logic processing will occur on the host side. Game data including the state of sprites on screen will be sent to the client. Thus, the client is merely reproducing on screen what the host is processing, and does no processing of its own other than sprite rendering.

A master FSM will control the state of the game, and the modes of gameplay (one- or two-player). The game logic processes the positions of sprites, and detects collisions. This game data is collected and transmitted to the network system, where data is packetized and sent over the serial line of communication. While game data is not being sent, voice data is packetized and transmitted, to enable voice communication between the two players.

2 Description

This section discusses each main component of the system, as well as the details of each component’s modules. Design decisions and implementations are discussed. The capacities of the components are explained, as well as their constraints. All modules are clocked at a frequency of 64.8MHz, produced by the FPGA’s Digital Clock Manager, based off a frequency of 27MHz. Figure 1 depicts a high-level block diagram of the overall system.

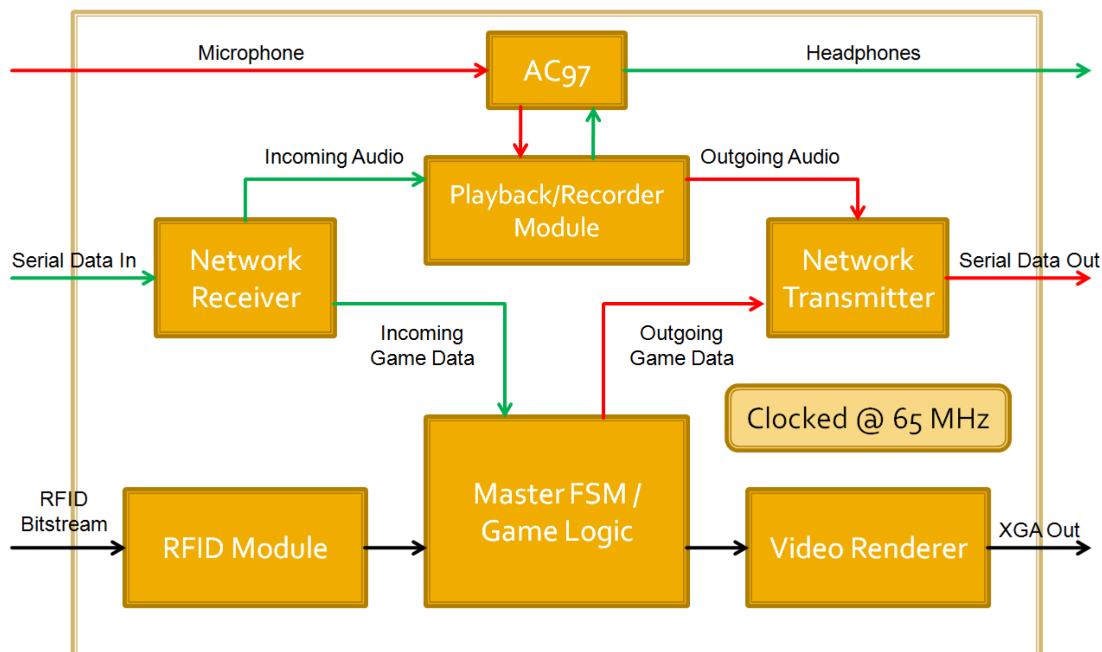

Figure 1: Top-Level Block Diagram

2.1 Master Finite State Machine (Javier)

The Master FSM module is a fairly simple but crucial component of the Game Controller and the system as a whole. This module oversees and manages the overall state of the system. Based on the user inputs it receives, it decides to either stay in the same state or change to another state. Most of the modules in the system depend on the "master" state to decide what and when to perform specific tasks. For example, the Pixel Generator decides what to render on the screen based on the "master" state. Although the Master FSM only outputs $state$ (4-bit bus), it receives nine inputs:

- 64.8 Mhz Clock: clock required to render XVGA video.
- Reset button: "enter" button on the labkit.
- Enter button: "enter" button on external PS/2 keyboard.
- One_player button: "1" button on external PS/2 keyboard.
- Two_players button: "2" button on external PS/2 keyboard.
- $Game_over$ signal: 1-bit signal from the Game Logic module.
- $External_state$ bus: 4-bit bus from the Network Controller which is originally transmitted by the other terminal.
- ID bus: 2-bit bus from switch 0 and switch 1 on the labkit.
- Win signal: 1-bit signal from the Game Logic module.

As illustrated by Figure 2, the FSM has three main branches the state of the game can follow: One Player Game, Host Two Player Game , and Client Two Player Game. The FSM initializes in the START state which asks the user to log in by using a valid "RFID tag" which we simulated with switch 0 (least significant bit) and switch 1. Using the switches, instead of a 10-byte ID number, we only had a 2-bit ID number. When the user inputs an appropriate ID, the FSM changes its state to the LOGIN state. On this screen, the user selects whether he or she wants to play a 1-player game or a 2-player game. If the user chooses to play a 1-player game, the FSM changes its state to ONE_PLAYER. The user presses enter when he or she is ready and the state changes to ONE_PLAYER_GAME; the game starts thereafter. During a 1-player game, there is no communication of game data between the two FPGA's; only audio is transmitted.

On the other hand, if the user chooses to play a 2-player game, the state changes to TWO_PLAYERS. The state that follows determines whether the terminal becomes the host or the client. When the user is ready, he or she presses enter and the state changes to WAITING_FOR_CLIENT – the terminal becomes the host. However, if the other user presses enter before, the state transitions to HOST_READY – the terminal becomes the client. Once in the HOST_READY state, the user presses enter to start the game; the client terminal transitions to the TWO_PLAYER_CLIENT state while the host terminal transitions to the TWO_PLAYER_HOST state.

To accomplish the behavior described above, the FSM module has two main components:

1. The “state” combinational logic takes charge of deciding what the next state will be. Since our FSM has 11 states, a 4-bit register is necessary to account for all of them. Therefore, the FSM module checks the current state by matching the 4-bit “state” register to a 4-bit value that corresponds to a state. Once it has identified the current state, it will check the values of the inputs and decide what the “next state” will be. Table 1 illustrates the input conditions necessary for each transition.
2. The sequential logic ensures that the state changes to the “next state” (determined in the “state” combinational logic) at every positive edge of the clock. Also, it changes the state to START when the Reset button is pressed.

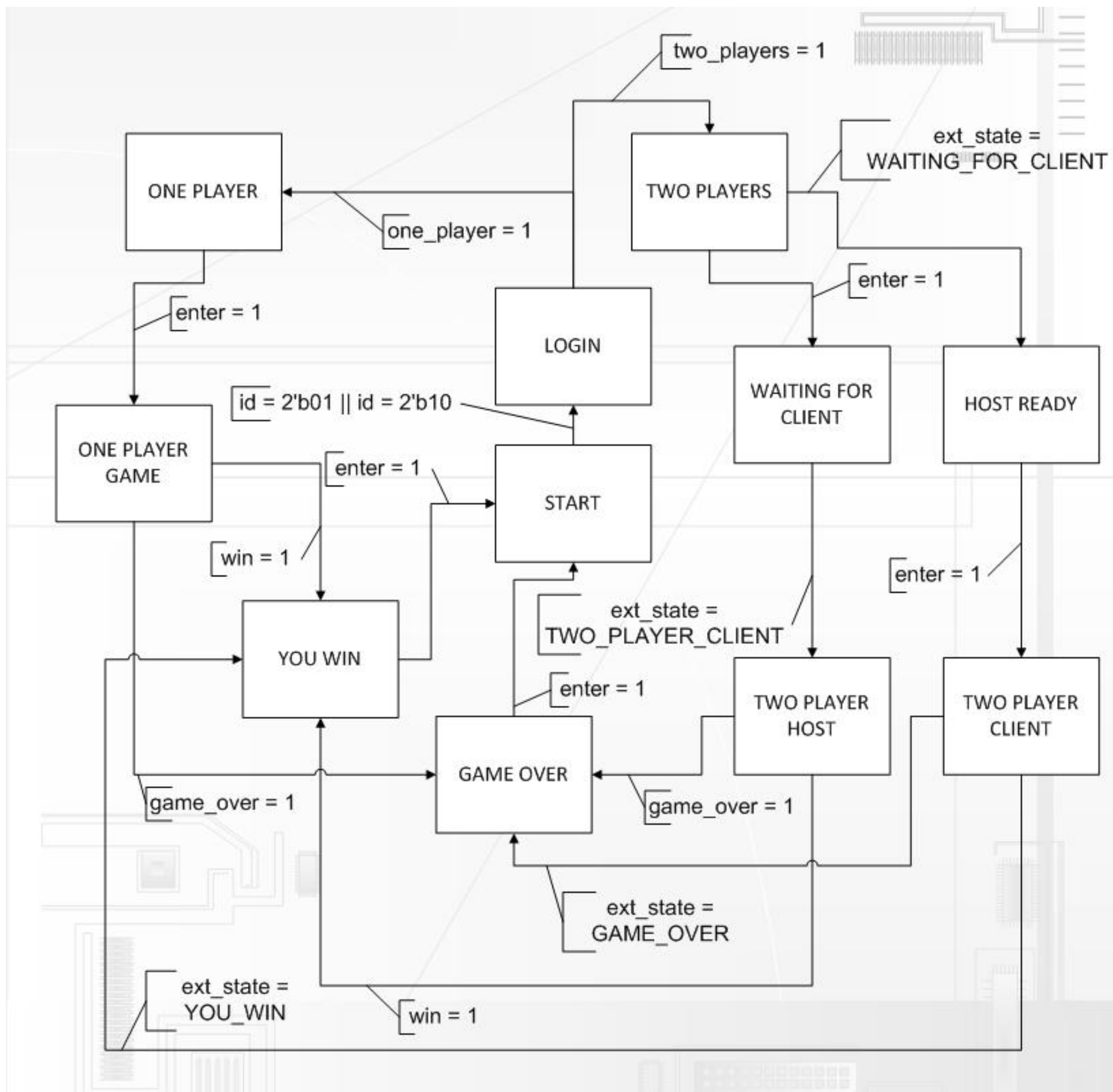

Figure 2: Master FSM Diagram

Table 1: FSM State Transitions

State	Condition 1	Next State	Condition 2	Next State
START	ID input matches one of the two ID numbers allowed to log into the system	LOGIN	N/A	N/A
LOGIN	one_player signal is HIGH	ONE_PLAYER	$two_players$ signal is HIGH	TWO_PLAYERS
ONE_PLAYER	Enter button is pressed	ONE_PLAYER_GAME	N/A	N/A
ONE_PLAYER_GAME	$game_over$ signal is HIGH	GAME_OVER	win signal is HIGH	YOU_WIN
TWO_PLAYERS	Enter button is pressed	WAITING_FOR_CLIENT	ext_state is WAITING_FOR_CLIENT	HOST_READY
HOST_READY	Enter button is pressed	TWO_PLAYER_CLIENT	N/A	N/A
WAITING_FOR_CLIENT	ext_state is TWO_PLAYER_CLIENT	TWO_PLAYER_HOST	N/A	N/A
TWO_PLAYER_HOST	$game_over$ signal is HIGH	GAME_OVER	win signal is HIGH	YOU_WIN
TWO_PLAYER_CLIENT	ext_state is GAME_OVER	GAME_OVER	ext_state is YOU_WIN	YOU_WIN
GAME_OVER	Enter button is pressed	START	N/A	N/A
YOU_WIN	Enter button is pressed	START	N/A	N/A

2.2 Game Logic (Javier)

The Game Logic module controls all the sprites displayed on the screen when the system is in “game” mode. The sprites represent player ships, enemy ships, and their bullets correspondingly. The movement of the player ships and the rate at which the bullets are fired is dependent on the user inputs. On the other hand, the module uses very simple Game Artificial Intelligence to determine the movement of the non-player characters (enemy ships) and their bullets. The main purpose of the Game Logic is to ensure that the users and the ships they control adhere to the strict rules of the game. Furthermore, the Game Logic also keeps track of score, determines when ships are killed, and decides when the players have won the game.

In order to achieve its tasks, the Game Logic needs to determine many details regarding the state of the characters in the game. These details range from the coordinates of the ships and bullets to the players' scores. To facilitate the management of all the calculations and keep track of all the sprites' coordinates, the Game Logic consists of four main components: Player Logic, Enemy Logic, Packets Logic, and Collision Detection Logic.

2.2.1 Player Logic

The Player Logic is designed to determine the x and y coordinates of each player and their corresponding bullets, and provide them to the Pixel Generator Module and the Collision Detector. For a 1-player game, the Player Logic takes the buttons on the labkit as inputs that determine the movement of Player 1 (up, down, left, right) and its bullet (button 0). However, for a 2-player game, the Player Logic varies depending on whether the terminal is serving as the client or the host. If the terminal acts as host, it functions much like the 1-player mode – labkit buttons control Player 1's movements and shooting. Meanwhile, input packets sent over the network from the client terminal serve as inputs that determine the movement of Player 2 and when it shoots. On the other hand, when the terminal functions as the client, there is no Player Logic because it is all done by the host.

Since, the player characters are allowed to move freely around the whole screen, the only movement limitations the Player Logic implements is with regard to the boundaries of the display screen. Therefore, using the inputs described above, the player spaceships move up, down, left, and right but do not go off the screen. The players move four pixels per frame in the direction of the button(s) pressed (up or down, and left or right). Because the two players have same limitations and follow the same rules, the only difference between the two is where their coordinates are initialized. Meanwhile, only one bullet per player is allowed on the screen, so the player cannot shoot until the bullet is off-screen; bullets go off-screen when they kill an enemy ship or when they go past the top boundary of the screen. The bullets move up 12 pixels per frame and they are fired whenever the button 0 is pressed and the bullet is off-screen.

2.2.2 Enemy Logic

The Enemy Logic is designed to determine the x and y coordinates of each non-player character (enemy spaceship) and their corresponding bullets, and provide them to the Pixel Generator Module and the Collision Detection Logic. The enemy Logic consists of three individual components, one for each of the enemy ships. However, besides having different initial coordinates, Enemy Logic 1 and Enemy Logic 2 are identical; therefore, we will discuss the two simultaneously and discuss Enemy Logic 3 separately.

Once again, the spaceships are allowed to move freely about the whole screen; however, unlike the players, the enemies can go off the screen. Enemy 1 and Enemy 2 move at a constant rate of 1 pixel per frame vertically and horizontally, but a vector $e12_x_movement$ controls the directions of their horizontal movement. When $e12_x_movement$ is HIGH, they move to the right; they move to the left when $e12_x_movement$ is LOW. The vector $e12_x_movement$

changes every 3 seconds (3X60 frames = 180 frames). Once they both leave the screen (move past the screen's bottom boundary), a counter $e12_respawn_counter$ starts counting. Enemy1 and Enemy2 respawn when the counter reaches 179; thus, they respawn 3 seconds after they die or move off the screen. Meanwhile, although their bullets also move 12 pixels per frame and follow the same restrictions as the player bullets – one bullet per ship – they are shot again as soon as they leave the screen so that they are constantly being fired.

In the meantime, although Enemy Logic 3 follows the same general pattern, there are some key differences. Enemy 3 not only moves faster (2 pixels per frame) but it also changes its horizontal direction more frequently (every 1.5 seconds), yet, its respawn rate of 4 seconds is slower than that of Enemy 1 and Enemy 2.

2.2.3 Packets Logic

The purpose of the Packets Logic is to ensure that the data from the "game" and "inputs" buffers received are assigned to the appropriate registers and that the appropriate values are assigned to the outgoing game and inputs buffers. The actions that the Packets Logic takes regarding incoming and outgoing data depend purely on the "master" state of the system. For example, if the terminal is the client during a 2-player game (state = TWO_PLAYER_CLIENT), the Packets Logic will do the following:

- Assign the incoming "game" buffer to the outgoing game buffer.
- Ignore the incoming "inputs" buffer.
- Assign the button inputs (up, down, left, right, and zero) to the outgoing inputs buffer.
- The outgoing "game" buffer is then sent to the Pixel Generator Module and the outgoing "inputs" buffer is sent to the host terminal.

Meanwhile, the host terminal would do almost the opposite:

- Ignore the incoming "game" buffer.
- Assign the bits from the incoming "inputs" buffer to the registers that correspond to the client's up, down, left, right, and enter which control the movement and shooting of Player 2.
- Assign the Game Logic registers (x and y coordinates, and the "display" bit) to the appropriate bit addresses of the outgoing "game" buffers.
- Assign LOW to the outgoing "inputs" buffer.

2.2.4 Collision Detection Logic

The Collision Detection Logic can be separated into three main groups: Player Collisions with Enemies, Enemy Bullet Collisions with Players, and Player Bullet Collisions with Enemies. The Collision Detection Logic takes the x and y coordinates of the objects in question and outputs a "display" bit which indicates whether the Player ship is touching the Enemy ship. All

ships are assumed to be squares with a set width and height and all bullets are circles with radius 2, except for Enemy 3 whose bullet has a radius of 4. With the heights and widths in mind, we carefully implemented a series of conditional statements that included multiple inequalities and ranges which allowed us to know whether the two squares (ships) in question are touching each other. When the Player ship collides with an Enemy ship the "display" signal is LOW indicating the Player has died. Similarly, if a bullet is found to collide with a ship, the "display" bit of that ship will be low, indicating it has been killed. For instance, if the "display" bit for both players is LOW, $game_over$ will change to HIGH.

2.3 Video Controller (Javier)

The Video Controller is responsible of generating the actual video signals that are sent to the XVGA output. The Video Controller includes the main Pixel Generator module which parses the data buffers into the appropriate registers and decides what pixels to display according to the state of the system. Moreover, the Video Controller also includes the Ship Sprites, Bullet Sprites, and Character String Display sub modules which are responsible for generating all objects on the screen based on their x and y coordinates and the "display" signal. Because the Ship Sprites module implements a three-stage pipeline, the bullet pixels and the sync and blank signals that are sent to the xvga display module (phsync, pvsync and pblank) are pipelined accordingly.

2.3.1 Pixel Generator Module

The Pixel Generator module parses the game data buffers received from the Game Logic Module into x and y coordinate registers and "display" registers for each ship and each bullet. It also decides which pixels to display. For instance, when the current state of the FSM is LOGIN, the Pixel Generator only displays strings of characters; however, if the FSM is in one of the TWO_PLAYER game states, the Pixel Generator will output pixels of all ships, bullets and strings displaying the score and name of each player.

2.3.2 Ship Sprites Modules

The Ships Sprites Modules are actually 5 different sprite modules which have the same format but display different images. Each corresponds to one of the Player or Enemy ships. We created the Player ships using Paint and saved them as 16-COLOR BITMAP files. Figure 3 shows a copy of the final designs of the Player Ships. Meanwhile, the Enemy ships were taken from a picture of a collection of alien spaceship designs (http://www.3drt.com/). The original image was cropped into three different images, each representing an Enemy ship. Finally, they were edited and saved as 16-COLOR BITMAP files. Figure 4 includes the original image and the three final designs.

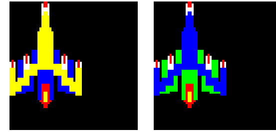

Figure 3: Player 1 (left) and Player 2 (right)

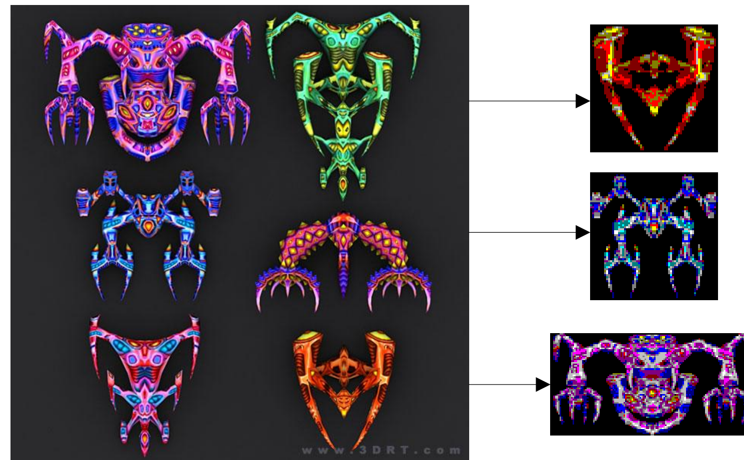

Figure 4: Enemy 1 (Top), Enemy 2 (Middle), and Enemy 3 (Bottom). Original Picture is on the left (downloaded from http://www.3drt.com/)

In order to create the images on the FPGA, we had to create Xilinx .COE files for each. We used the Matlab script BMPtoCOE.m to convert the 16-COLOR BITMAP files to .COE files. Then, we were able to use the FPGA's Single-Port block rams (BRAMs) to store the images. All ships except for Enemy 3 were 64 pixels by 64 pixels; Enemy 3 was 128 pixels by 64 pixels.

Because computing the read address takes two clock cycles and the BRAM access time accounts for an additional clock cycle, we had to use a three-stage pipeline. To create the BRAM address, the Ship Sprite module computes vertical and horizontal coordinates with respect to the graphic. The vertical coordinate is represented by $(vcount - ship_y)$ and the horizontal coordinate is $(hcount - ship_x)$, where $ship_x$ and $ship_y$ are the coordinates of the top-left hand side of the graphic. Normally, the read-address is obtained by multiplying $(vcount - ship_y)$ by the width of the image and then adding the product to the value of $(hcount - ship_x)$. However, we purposely chose the sizes of the images to be powers of two in order to avoid multiplying by shifting instead. To clarify, let us demonstrate an example of how we calculated the BRAM address:

- $(vcount - ship_y)$ computes how far into the picture vcount is with respect to the top of the ship.
- $(hcount - ship_x)$ computes how far into the picture hcount is with respect to the left of the ship.
- Finally, we can get the read address:

$$raddr \leq \{ship_addr[5:0]\,, ship_hcount[5:0]\ \};$$

Finally, because we use 3-bit RGB values for our game display, we had to map the 4-bit colors to a corresponding 3 bit-color. I used the values in Table 2.

Table 2: 4-bit to 3-bit Color Conversion

Color	3-bit Color (Binary RGB)	4-bit Colors (Decimal)
Black	000	0
		7
White	111	8
		15
Blue	001	4
		12
Green	010	2
		10
Light Blue	011	6
		14
Red	100	1
		9
Pink	101	5
		13
Yellow	110	3
		11

2.3.3 Bullet Sprites Modules

The Bullet Sprites Module outputs color pixels that create the shape of a circle as long as the "display" bit is HIGH. Because computing the distance formula takes two clock cycles, we have to implement a two-stage pipeline to stay synchronized with the other graphics. This module has only three inputs:

- "Display" bit.
- X coordinates of the bullet.
- Y coordinates of the bullet.

The bullets of Player 1 and Player 2 are white (3'b111) and have a 2-pixel radius. Similarly, the bullets of Enemy 1 and Enemy 2 also have a 2-pixel radius but they are pink (3'b101) instead of white. Nevertheless, the bullet of Enemy three has a 4-pixel radius and is red (3'b100).

2.3.4 Character String Display Module

The Character String Display sub-modules use the sample code cstringdisp.v written by Chris Terman and I. Chuang to display the character strings of the game. The Start and Game

Over screens along with their corresponding "Press Enter" messages are displayed by this module. Also, the name, ID number and score of each player is also displayed by these modules.

2.4 Audio Controller (Charlie)

This section is organized by the flow of information the data takes. Voice data is originated at the Audio Recorder module, passed on to the Record Buffer, received by the Play Buffer, and finally arrives at the Audio Player for processing. Figure 5 depicts the data pathways:

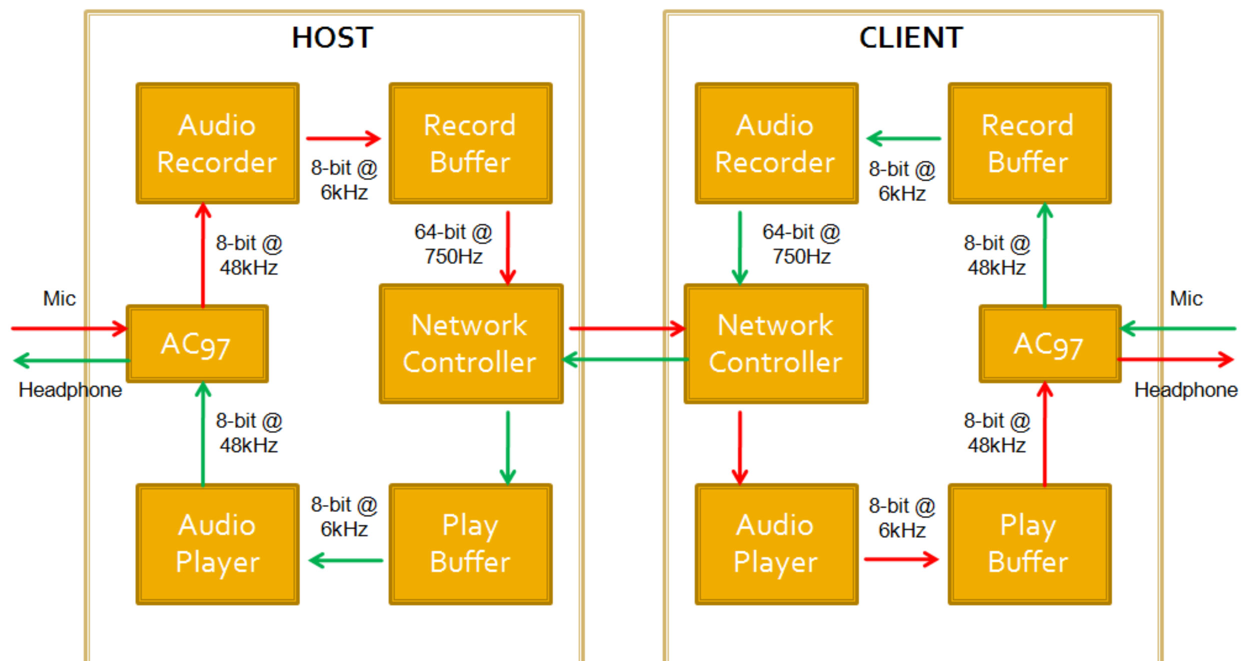

Figure 5: Audio Controller Block Diagram

2.4.1 Audio Recorder

The Audio Recorder module takes 8-bit PCM audio samples from the AC97 microphone port, *from_ac97_data*, at a rate of 48 kHz. These samples are passed through a 3 kHz low-pass filter, which will compensate for the sample aliasing that occurs when the audio is downsampled to a frequency of 6 kHz. The module accomplishes this downsampling by keeping a count of the incoming samples, and outputting every 8th filtered sample. A "recording sample ready" output signal (*R_sample_ready*) goes high for one clock cycle when a new sample is available at the 8-bit *R_audio_out_data* output bus.

In effect, the Audio Recorder module takes in audio samples at a rate of 48 kHz, and outputs filtered samples at a rate of 6 kHz. A "ready" signal from the AC97 queues the module to take in a new audio sample.

2.4.2 Record Buffer

The Record Buffer accepts filtered, downsampled audio samples from the Audio Recorder module, at a rate of 6 kHz. The buffer is an array of eight 8-bit registers, and is filled sequentially as new samples arrive. A new sample is latched from the *R_audio_in_data* 8-bit bus when the "recording sample ready" signal (*R_sample_ready*) goes high, and stored in the buffer. Upon arrival of the 8th sample, the contents of the entire buffer are output to a 64-bit bus, *R_buffer_out*, and a "ready" signal (*R_buffer_ready*) is pulsed for one-clock cycle. New samples overwrite the contents of the old buffer.

Essentially the Record Buffer accepts individual audio samples and collects them to output a set of 8 samples simultaneously, at a rate of 750 Hz. This buffer is the content of the audio packet that is transmitted over the network, and is made available once every 86,400 clock cycles.

2.4.3 Play Buffer

The Play Buffer module takes as input a 64-bit audio packet, containing eight 8-bit audio samples, to be played back at a rate of 6 kHz. A new audio packet is available at the 64-bit bus *P_buffer_in* when the "playback buffer ready" signal (*P_buffer_ready*) is high. Upon receiving a packet, individual audio samples are latched to an array of eight 8-bit registers. The module generates a one-clock pulse signal at a rate of 6 kHz. When this signal is high, the "playback sample ready" signal (*P_sample_ready*) is pulsed as well, and a new audio sample is made available at the 8-bit *P_audio_out_data* port. Buffered samples are played in sequence, and when all samples are played, the last sample in the buffer is held to the output until a new packet arrives. The effect of this is that if a new packet does not arrive on time, the last sample received is played over and over again.

2.4.4 Audio Player

The Audio Player is designed to receive audio samples at a rate of 6 kHz, and passes them through a low-pass reconstruction filter to get rid of audio artifacts that arise from upsampling the audio to 48 kHz. The module receives a "ready" signal (*audio_ready*) from the AC97 at a rate of 48 kHz, and the module feeds an incoming audio sample from the 8-bit bus *P_audio_in_data* to the filter on every 8th trigger of the AC97's signal. The filter is otherwise fed a zero-expanded sample. Filtered samples are then latched to the AC97's headphone input, *to_ac97_data*, whenever the AC97's "ready" signal is high.

As such, this module will receive audio samples at a rate of 6 kHz, and output audio samples at a rate of 48 kHz, after proper filtering and upsampling.

2.5 Network Controller (Charlie)

As in the previous section, this section is organized by the flow of information. Game and voice data is received and packetized by the Network Transmitter, sent byte-by-byte to the Serializer, then transmitted bit-by-bit over the serial port to the De-Serializer where the data is

decoded, and finally buffered to the Network Receiver, where the final game data is parsed out to the rest of the system. Figure 6 depicts the data pathways:

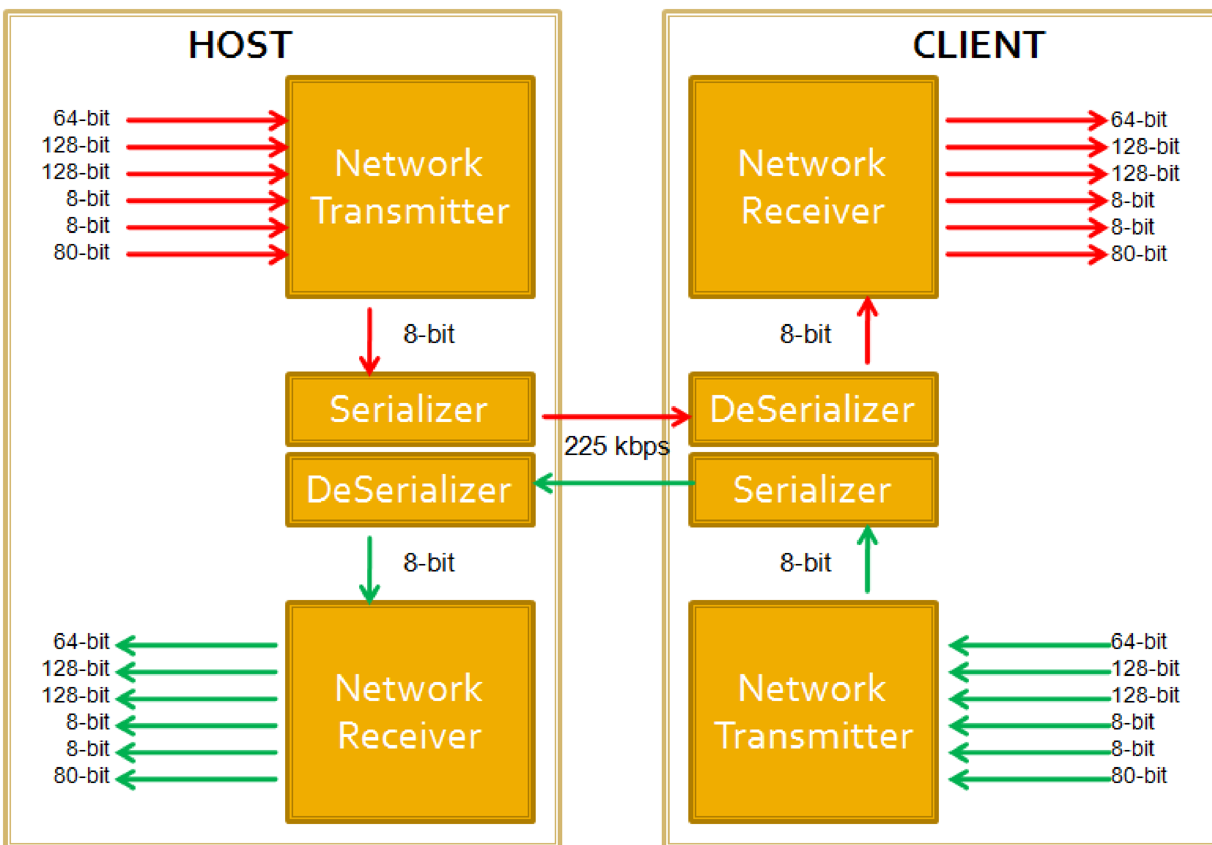

Figure 6: Network Controller Block Diagram

2.5.1 Network Transmitter

The Network Transmitter module handles all outbound communication in the form of packets of data. Packets can vary in size, measured in bytes. All packets assume the structure of a one-byte header followed by a variable size payload. The header is a unique combination of bits that allow the Receiver to determine what kind of data is contained in the packet. This implementation is specifically designed to be able to transmit six different kinds of packets, as specified in Table 3.

Table 3: Packet Types

Packet Header	Payload Type	Payload Size	Payload Contents
0b00000001 = 0x0001	Audio Data	64 bits = 8 bytes	Eight audio samples
0b00001000 = 0x0008	Game Data 0	128 bits = 16 bytes	Five sprite coordinates
0b00001001 = 0x0009	Game Data 1	128 bits = 16 bytes	Five sprite coordinates
0b00001010 = 0x000A	Game Data 2	8 bits = 1 byte	Game state
0b00001011 = 0x000B	Game Data 3	8 bits = 1 byte	Client user's input
0b00001100 = 0x000C	Game Data 4	80 bits = 10 bytes	User ID tag/string

Not all bits of the payload data in these packet structures are significant to the functions of the game. Figure 7 depicts the payload structures of each packet (LSB to MSB from left to right), as well as any unused bits.

Audio Payload

8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit
Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7	Sample 8

Game Data 0 Payload

11-bit	10-bit	1-bit	11-bit	…	11-bit	10-bit	1-bit	5-bit	5-bit	8-bit
Sprite 1 X-Coord.	Sprite 1 Y-Coord.	Sprite 1 Display Bit	Sprite 2 X-Coord.	…	Sprite 5 X-Coord.	Sprite 5 Y-Coord.	Sprite 5 Display Bit	Player 1 Score	Player 2 Score	Unused (Zeros)

Game Data 1 Payload

11-bit	10-bit	1-bit	11-bit	10-bit	1-bit	…	11-bit	10-bit	1-bit	18-bit
Sprite 6 X-Coord.	Sprite 6 Y-Coord.	Sprite 6 Display Bit	Sprite 7 X-Coord.	Sprite 7 Y-Coord.	Sprite 7 Display Bit	…	Sprite 10 X-Coord.	Sprite 10 Y-Coord.	Sprite 10 Display	Unused (Zeros)

Game Data 2 Payload

4-bit	4-bit
Master FSM State	Unused (Zeros

Game Data 3 Payload

bit 0	bit 1	bit 2	bit 3	bit 4
Button 0 (Fire)	Button Up	Button Down	Button Left	Button Right

Game Data 4 Payload

8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit	8-bit
ID Byte 0	ID Byte 1	ID Byte 2	ID Byte 3	ID Byte 4	ID Byte 5	ID Byte 6	ID Byte 7	ID Byte 8	ID Byte 9

Figure 7: Packet Structures

Audio packets are continuously sent by the Network Transmitter when triggered by the “ready” signal, *R_buffer_out*. This means audio packets are sent out once every 86,400 clock cycles. As will be detailed in the following section, the Serializer module is designed to operate at 225kbps. At this data rate, it takes 25,920 clock cycles to send an audio packet, which leaves plenty of time until the next audio packet needs to be transmitted. Game packets are sent on the rising edge of the XVGA Module’s *vsync* signal, in other words, at a rate of 60Hz, or once every 1,080,000 clock cycles. The combination of game packets that needs to be sent out in any given frame is determined by the state of the game. Theoretically, the maximum amount of time it can take to send game packets, that is, to send all five game packets in the frame, is

141,120 clock cycles. In this extreme scenario, at least one audio packet "send request" will be ignored by the Network Transmitter while game packet data is being sent. As such, we decided that this audio packet is best discarded, and the audible impact was not discernable. However, if the rising edge of *vsync* happens to occur while an audio packet is being transmitted through the network, the Network Transmitter will hold the game data until the serial line is idle, then proceed with sending the game data.

The combination of game packets that need to be sent in a given frame depends on the state of the game. Table 4 describes all possible combinations implemented in our design, the total size (including header byte) as well the time it takes (in clock cycles) to send.

Table 4: Packet Transmission Send Times

Combination						Total Size (bytes)	TX Time (cycles)
Audio	Game 0 (Coordinate)	Game 1 (Coordinate)	Game 2 (State)	Game 3 (Client Input)	Game 4 (ID Tag)		
			•			2	5,760
	•	•	•			36	103,680
			•	•		4	11,520
			•		•	13	37,440
•						9	25,920

2.5.2 Serializer

The Serializer module handles the lowest-level of communication between the two FPGA platforms. The Serializer uses two ports on the serial port, the TXD signal and the RTS signal. When idle, these two signals are high. The RTS signal indicates to the receiving end that the line is idle. The RTS signal is also wired to the *tx_ready* signal, which is fed to the Network Transmitter module, and indicates when the Serializer is ready to transmit data. The RXD line is operated using the RS-232 scheme, which means a byte of data is transmitted by sending a "start bit" (logic zero), followed by eight bits of data, and ending with one "stop bit" (logic one). For every byte of data sent, ten bits are actually transmitted. The Serializer operates the RXD line at a data rate of 225,000 bits per second. The actual hardware is theoretically capable of transmitting at a rate of 250 kbps, but we chose 225 kbps because a 225 kHz clock can be evenly counted by the 64.8 MHz system-wide clock. With this design, the Network Controller may not send or receive fractional parts of a byte of data.

The following formula can be used to calculate the time (in clock cycles) it takes to send data over the network:

$$t_{cycles} = \frac{(x_{bytes}) * \left(10 \frac{bits}{byte}\right)}{225000 \frac{bits}{second}} * \left(64800000 \frac{cycles}{second}\right)$$

When the Network Controller needs to send data, it pulses the *tx_request* signal high, and the Serializer latches the data from *symbol_tx* to an 8-bit buffer.

2.5.3 De-Serializer

The De-Serializer module receives asynchronous data from the serial port, decoding it according to the RS-232 scheme and outputting a byte of data at a time. It uses two signals of the serial port, the RXD signal and the CTS signal. These inputs are fed from the Serializer module on the other terminal. That is, the client's RXD signal is connected to the host's TXD output. Similarly, the client's CTS signal is tied to the host's RTS signal, and vice versa. Thus, RXD and CTS are high when idle. CTS is low when there is incoming data on the RXD line.

This module oversamples the RXD line at eight times the transmitting data rate, that is, at a frequency of 1.8 MHz. Spikes on the transmission line are filtered using flip-flops, and the line is synchronized to the De-Serializer clock when a start bit is detected. When the start bit is detected, data bits are latched at the predetermined rate of 225 kHz. When the 8-bit buffer, which is storing the incoming data, is full, the *rx_data_ready* signal is pulsed high, and the data held to the output *symbol_rx*.

This simple asynchronous serial data receiver is not designed for error checking, let alone error correction. The Serializer does not transmit a parity bit that the De-Serializer could then use to verify the integrity of the data. This design decision was largely made in part because the practical implications it could have caused due to extra network overhead could not be determined. Time constraints disallowed for the design of a more robust system.

2.5.4 Network Receiver

The Network Receiver handles all inbound communication, and parses incoming data to the appropriate buffers and system modules. When the Network Receiver module is idle, and the De-Serializer module's signal *rx_data_ready* signal is pulsed high, the Receiver latches the data at the *symbol_rx* bus, and reads it as a header byte. Depending on the value of the 8-bit header, the Receiver determines what kind of packet is being received. The type also determines how many bytes of data are incoming, and keeps track of how many have been received. When its buffer is filled, it outputs the data to the appropriate bus. If it's an audio packet, it will output the 8-byte data to the 64-bit bus *P_buffer_out* and pulse the "playback buffer ready" signal *P_buffer_ready*. If it's a game packet, the module determines what kind of game packet it is, and output it to the corresponding bus.

The limitations of this simple implementation are evident; the system cannot account for interrupted communication. The Receiver follows a finite state machine model, and is triggered by the *rx_data_ready* signal to change states in sequence. It does not check whether the incoming byte is payload data or the header byte of a totally new packet.

2.6 RFID Interface (Charlie)

Due to time constraints, the RFID interface could not be fully integrated into the final implementation of Space Force Duo. The RFID interface consists of a module that reads in data

from a Parallax RFID Card Reader with a serial interface. The Card Reader outputs data serially at a rate of 2400 bps. As such, interfacing with the card requires a mere adaptation of the Serializer module described previously. The Card Reader outputs 12 bytes of data when an RFID transponder is detected. The 12-byte sequence consists of a “Start Byte” (0x0A) followed by a unique sequence of a 10-byte ID, and ends with a “Stop Byte” (0x0D). Reading and buffering these bytes was also a simple adaptation of the Network Receiver module.

There were problems with the RFID module that could not be fully addressed given the time constraints. One issue was that the RFID tags would not output its unique ID consistently. There were infrequent occasions where the ID that the RFID module output was not the same as the tag’s ID. Because of this, we could not implement a reliable system of logging in to the Space Force Duo game. The RFID functionality was instead replaced with switch inputs corresponding to a unique ID.

3 Conclusion

3.1 Testing and Debugging

ModelSim was essential in implementing a proper communications system, allowing for the simulation of the behavior of signals, in a quick and efficient manner. Specifically, the timing of signals for the serializer modules was necessary in order to achieve consistent and reliable data transmission. Actual testing involved implementing a serializer on one labkit and a deserializer on the other labkit, connecting them with a serial cable, and then transferring a simple running count from one labkit to the other.

A more concrete test could be performed once the Audio Controller was complete. Once we obtained a consistent stream of pre-processed audio samples, we could establish two-way voice communication between the two labkits. Identifying artifacts in the audio was an easy way to identify bugs in the system.

Game packetization and transmission was also simulated with ModelSim. For an effective communications system, we had to ensure that packets were being transferred in the right order, at the right time.

Meanwhile, the Game Logic and Master FSM were tested mainly by using the XVGA display. The fluorescent displays, the switches, the buttons and the LED lights on the labkit were also useful when the screen would not display the appropriate graphics. For instance, to debug the state of the system and the coordinates of the Player and its bullet were displayed on the fluorescent displays.

When testing the two-player game before the networking had been developed, we tested by using the switches to simulate the state data from the “other” terminal and controlling the movement and shooting of both players by using the same buttons.

3.2 Results

The resulting final implementation turned out to be a challenging two-player cooperative game, though simplistic in nature. Though it was not without its bugs and minor glitches, it

successfully demonstrated the ability to transmit data back and forth in real time between the two FPGA platforms. The game logic is a highly complex system, handling multiple sprites on screen, detecting collisions, and directing the enemy AI. For the one-player game, the logic performed exceptionally well, looking like nothing less than a classic arcade game. When put to the test on the network two-player mode, the game performed adequately, showing only minor bugs whose sources were hard to detect. Given the time constraints, we were unable to fully determine whether these glitches were due to the game logic, or network transmission issues. Some of these glitches include players dying unexpectedly, enemy ships not firing bullets, and scores not being counted correctly. Voice communication worked flawlessly. As predicted, losing a few audio packets here and there had negligible impact.

The RFID feature of logging into the game was only absent because of time constraints, not because of shear difficulty in implementing it. As demonstrated, we were able to interface with the card reader, and obtain more or less consistent tag ID's from the different transponders. However, we decided that this feature would not have as impressive of an impact as, say, a more complex video game, or a reliable communication system. Hence we did not dedicate much time towards implementing this feature.

Many improvements could have been made to this project. For instance, a more robust serial communication system could have handled packet transactions better, to prevent data loss and perform error correction on data. One simple data integrity check could be to transmit a parity bit on every byte of data sent, and have the serializer modules check this bit. A false parity test result could deem the entire packet as corrupt, and discard it as such. A more functional and detailed header could incorporate such descriptions as the length of the packet payload, or the CRC of the payload data, for error detection. On a higher level, a packet acknowledgement system could also be implemented, where a short acknowledgement packet could be sent by the receiver to the sender when a good packet has arrived. The sender could keep track of a time-out counter, whereby if a packet is not acknowledged before the expiration of the timer, the packet could be resent. Had these ideas been implemented into the communication system, we may have run into issues where too many packets are being sent back and forth within frames. As calculated before, rendering a XVGA frame at 60 Hz gives us about one million clock cycles to complete all packet operations for the frame. If we needed more than this, could have easily lowered the frame rate to 30 Hz, and that would grant us about two million cycles instead.

A FIFO packet queuing system could eliminate packet loss, as we had with the audio packets. This would probably be a necessary tool if any of the previous ideas were to be implemented as well.

Overall, however, our basic assumptions and design decisions resulted in a well-implemented video game with much room for improvement. We feel confident having achieved our goals for this project with our final implementation of Space Force Duo.

4 References

Serial Interface. *Fpga4fun.com*, http://www.fpga4fun.com/SerialInterface.html (12/10/09)

Appendix: Verilog Code

A. Top Level Module

```
///////////////////////////////////////////////////////////////////////////////
//
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
```

```
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////
//

      module Final( / * beep, * / audio_reset_b, ac97_sdata_out,
ac97_sdata_in, ac97_synch,
      ac97_bit_clock,

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

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

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

//             ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
//             ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,
//
//             ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
//             ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,
//
//             clock_feedback_out, clock_feedback_in,
//
//             flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
//             flash_reset_b, flash_sts, flash_byte_b,
//
      rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

//             mouse_clock, mouse_data,

      keyboard_clock, keyboard_data,

      clock_27mhz, clock1, clock2,

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

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

      switch,
```

```
      led,

//           user1, user2, user3, user4,

//           daughtercard,

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

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

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

output[7:0]vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
      vga_out_hsync, vga_out_vsync;
//
//   output [9:0] tv_out_ycrcb;
//   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
//      tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
//      tv_out_subcar_reset;
//
//   input  [19:0] tv_in_ycrcb;
//   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
tv_in_aef,
//      tv_in_hff, tv_in_aff;
//   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
//      tv_in_reset_b, tv_in_clock;
//   inout  tv_in_i2c_data;

//   inout  [35:0] ram0_data;
//   output [18:0] ram0_address;
//   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;
//   output [3:0] ram0_bwe_b;
//
//   inout  [35:0] ram1_data;
//   output [18:0] ram1_address;
//   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;
//   output [3:0] ram1_bwe_b;
//
//   input  clock_feedback_in;
//   output clock_feedback_out;
//
//   inout  [15:0] flash_data;
//   output [23:0] flash_address;
//   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
//   input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;
```

```
input / * mouse_clock, mouse_data, * / keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

//   inout [43:0] daughtercard;

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

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


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

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

   // Audio Input and Output
   //assign beep= 1'b0;
   //assign audio_reset_b = 1'b0;
   //assign ac97_synch = 1'b0;
   //assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // Video Output
//   assign tv_out_ycrcb = 10'h0;
//   assign tv_out_reset_b = 1'b0;
//   assign tv_out_clock = 1'b0;
//   assign tv_out_i2c_clock = 1'b0;
//   assign tv_out_i2c_data = 1'b0;
//   assign tv_out_pal_ntsc = 1'b0;
//   assign tv_out_hsync_b = 1'b1;
//   assign tv_out_vsync_b = 1'b1;
//   assign tv_out_blank_b = 1'b1;
//   assign tv_out_subcar_reset = 1'b0;
//
//   // Video Input
//   assign tv_in_i2c_clock = 1'b0;
//   assign tv_in_fifo_read = 1'b0;
```

```
//   assign tv_in_fifo_clock = 1'b0;
//   assign tv_in_iso = 1'b0;
//   assign tv_in_reset_b = 1'b0;
//   assign tv_in_clock = 1'b0;
//   assign tv_in_i2c_data = 1'bZ;
//   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
//   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs
//
//   // SRAMs
//   assign ram0_data = 36'hZ;
//   assign ram0_address = 19'h0;
//   assign ram0_adv_ld = 1'b0;
//   assign ram0_clk = 1'b0;
//   assign ram0_cen_b = 1'b1;
//   assign ram0_ce_b = 1'b1;
//   assign ram0_oe_b = 1'b1;
//   assign ram0_we_b = 1'b1;
//   assign ram0_bwe_b = 4'hF;
//   assign ram1_data = 36'hZ;
//   assign ram1_address = 19'h0;
//   assign ram1_adv_ld = 1'b0;
//   assign ram1_clk = 1'b0;
//   assign ram1_cen_b = 1'b1;
//   assign ram1_ce_b = 1'b1;
//   assign ram1_oe_b = 1'b1;
//   assign ram1_we_b = 1'b1;
//   assign ram1_bwe_b = 4'hF;
//   assign clock_feedback_out = 1'b0;
//   // clock_feedback_in is an input
//
//   // Flash ROM
//   assign flash_data = 16'hZ;
//   assign flash_address = 24'h0;
//   assign flash_ce_b = 1'b1;
//   assign flash_oe_b = 1'b1;
//   assign flash_we_b = 1'b1;
//   assign flash_reset_b = 1'b0;
//   assign flash_byte_b = 1'b1;
//   // flash_sts is an input

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

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays
//   assign disp_blank = 1'b1;
//   assign disp_clock = 1'b0;
//   assign disp_rs = 1'b0;
//   assign disp_ce_b = 1'b1;
//   assign disp_reset_b = 1'b0;
//   assign disp_data_out = 1'b0;
   // disp_data_in is an input
```

```
   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
//   assign user1 = 32'hZ;
//   assign user2 = 32'hZ;
//   assign user3 = 32'hZ;
//   assign user4 = 32'hZ;
//
//   // Daughtercard Connectors
//   assign daughtercard = 44'hZ;
//
//   // SystemACE Microprocessor Port
//   assign systemace_data = 16'hZ;
//   assign systemace_address = 7'h0;
//   assign systemace_ce_b = 1'b1;
//   assign systemace_we_b = 1'b1;
//   assign systemace_oe_b = 1'b1;
//   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;


////////////////////////////////////////////////////////////////////////////
   //
   // Final: final project (Space Force Duo
   //

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

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf, clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz), .CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz), .I(clock_65mhz_unbuf));

   // power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr(.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
      .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
```

```
wire reset, user_reset;
debounce db1(.reset(power_on_reset), .clock(clock_65mhz), .noisy(
~button_enter), .clean(user_reset));
assign reset = user_reset | power_on_reset;

   // UP and DOWN buttons for pong paddle
wire up_button, down_button, left_button, right_button, button_0, button_1,
button_2, button_3;
debounce db2(.reset(reset), .clock(clock_65mhz), .noisy( ~button_up),
.clean(up_button));
debounce db3(.reset(reset), .clock(clock_65mhz), .noisy( ~button_down),
.clean(down_button));
debounce db4(.reset(reset), .clock(clock_65mhz), .noisy( ~button_left),
.clean(left_button));
debounce db5(.reset(reset), .clock(clock_65mhz), .noisy( ~button_right),
.clean(right_button));
debounce db6(.reset(reset), .clock(clock_65mhz), .noisy( ~button0),
.clean(button_0));
debounce db7(.reset(reset), .clock(clock_65mhz), .noisy( ~button1),
.clean(button_1));
debounce db8(.reset(reset), .clock(clock_65mhz), .noisy( ~button2),
.clean(button_2));
debounce db9(.reset(reset), .clock(clock_65mhz), .noisy( ~button3),
.clean(button_3));

   // generate basic XVGA video signals
wire[10:0] hcount;
wire[9:0] vcount;
wire hsync, vsync, blank;
xvga xvga1(.vclock(clock_65mhz), .hcount(hcount), .vcount(vcount),
      .hsync(hsync), .vsync(vsync), .blank(blank));

   // feed XVGA signals to user's pong game
wire[2:0] pixel;
wire phsync, pvsync, pblank;




      // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
reg[2:0] rgb;
reg b, hs, vs;
always @(posedge clock_65mhz) begin
         // default: pixel generator
      hs <= phsync;
      vs <= pvsync;
      b <= pblank;
      rgb <= pixel;
     // end
end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
assign vga_out_red = {8{rgb[2]}};
```

```
assign vga_out_green = {8{rgb[1]}};
assign vga_out_blue = {8{rgb[0]}};
assign vga_out_sync_b = 1'b1;    // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

//  assign led = ~{3'b000,up,down,reset,switch[1:0]};


wire[63:0] counter_disp;
wire e1_bullet_disp;
wire e2_bullet_disp;
wire e12_x_movement;
wire[3:0] state;
wire game_over;
wire[127:0] game_packet1;
wire[127:0] game_packet2;
reg enter;
reg one_player;
reg two_players;
wire[7:0] ascii;
wire ascii_ready;


wire[127:0] GTX_buffer0, GTX_buffer1, GRX_buffer0, GRX_buffer1;
wire[7:0] GRX_buffer2, GRX_buffer3, GRX_buffer4;
wire[3:0] GTX_buffer2;
wire[4:0] GTX_buffer3;
wire[1:0] GTX_buffer4;

assign counter_disp[23:0] = GTX_buffer0[20:0];
assign counter_disp[47:24] = GTX_buffer1[20:0];
assign counter_disp[55:48] = GTX_buffer0[76:66];
assign counter_disp[59:56] = GRX_buffer2;
assign counter_disp[63:60] = GTX_buffer2;


//FSM inputs from keyboard
always @(posedge clock_65mhz) begin
      if (ascii_ready) begin
            if (ascii == 8'h0D) begin
                  enter <= 1;
                  one_player <= 0;
                  two_players <= 0;
            end
            else if (ascii == 8'h31) begin
                  enter <= 0;
                  one_player <= 1;
                  two_players <= 0;
            end
            else if (ascii == 8'h32) begin
                  enter <= 0;
                  one_player <= 0;
                  two_players <= 1;
            end
```

```
            else begin
                  enter <= 0;
                  one_player <= 0;
                  two_players <= 0;
            end
      end
      else begin
            enter <= 0;
            one_player <= 0;
            two_players <= 0;
      end
end


wire[4:0] p1_score;
wire[4:0] p2_score;
wire win;
assign GTX_buffer4 = switch[1:0];

ps2_ascii_input keyboard(.clock_65mhz(clock_65mhz), .reset(reset),
.clock(keyboard_clock), .data(keyboard_data),
      .ascii(ascii), .ascii_ready(ascii_ready));

display_16hex display(.reset(reset), .clock_65mhz(clock_65mhz),
.data(counter_disp),
      .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
.disp_ce_b(disp_ce_b),
      .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));

fsm master_fsm(.clk(clock_65mhz), .reset(reset), .enter(enter),
      .one_player(one_player), .two_players(two_players),
.game_over(game_over), .win(win), .ext_state(GRX_buffer2[3:0]),
.id(GTX_buffer4),
      .state(GTX_buffer2));

pixel_gen generator(.vclock(clock_65mhz), .reset(reset),
      .hcount(hcount), .vcount(vcount), .hsync(hsync), .vsync(vsync),
.blank(blank),
      .game_packet1_in(GTX_buffer0), .game_packet2_in(GTX_buffer1),
      .state(GTX_buffer2), .id_packet_in(GRX_buffer4), .id(GTX_buffer4),
.p1_score(p1_score), .p2_score(p2_score),
      .phsync(phsync), .pvsync(pvsync), .pblank(pblank), .pixel(pixel) / * ,
.score1_cstring(counter_disp[23:0]) * / );


game_logic game(.vclock(clock_65mhz), .reset(reset),
      .up(up_button), .down(down_button), .left(left_button),
.right(right_button), .button_0(button_0),
      .hsync(hsync), .vsync(vsync), .blank(blank),
      .state_packet_in(GTX_buffer2), .inputs_packet_in(GRX_buffer3[4:0]),
      .game_packet1_in(GRX_buffer0), .game_packet2_in(GRX_buffer1),
      .game_packet1_out(GTX_buffer0), .game_packet2_out(GTX_buffer1),
.game_over(game_over), .win(win),
      .p1_score(p1_score), .p2_score(p2_score),
.inputs_packet_out(GTX_buffer3));

//debugging
```

```
assign led[0] = ~e1_bullet_disp;
assign led[1] = ~e2_bullet_disp;
assign led[2] = ~GTX_buffer0[21]; //player1_display
assign led[3] = ~GTX_buffer1[21];//p1_bullet_display
assign led[4] = ~GTX_buffer0[43]; //enemy1_display
assign led[5] = ~GTX_buffer1[43];//e1_bullet_display
assign led[6] = ~GTX_buffer0[65]; //enemy2_display
assign led[7] = ~GTX_buffer1[65];//e2_bullet_display


      //***NETWORK***//

wire[7:0] from_ac97_data, to_ac97_data;
wire audio_ready;

   // allow user to adjust volume
reg old_vup, old_vdown;
reg[4:0] volume;
always @(posedge clock_65mhz) begin
      if (reset)volume <= 5'd8;
      else begin
            if (button_2 & ~old_vup & volume != 5'd31)volume <= volume + 1;
            if (button_3 & ~old_vdown & volume != 5'd0)volume <= volume - 1;
      end
      old_vup <= button_2;
      old_vdown <= button_3;
end

wire[7:0] R_audio_out_data, P_audio_out_data;
wire R_sample_ready, R_buffer_ready, P_sample_ready, P_buffer_ready;
wire[63:0] R_buffer_out, P_buffer_out;


wire[7:0] symbol_tx, symbol_rx;

wire tx_ready, line, tx_request;

      // AC97 Interface
audio a(clock_65mhz, reset, volume, from_ac97_data, to_ac97_data,
audio_ready,
      audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
ac97_bit_clock);

      // Recorder Module
recorder audio_record(.clock(clock_65mhz), .reset(reset),
.audio_ready(audio_ready),
      .from_ac97_data(from_ac97_data), .R_audio_out_data(R_audio_out_data),
.R_sample_ready(R_sample_ready));

      // Recorder Buffer
record_buffer audio_record_buffer(.clock(clock_65mhz), .reset(reset),
.R_sample_ready(R_sample_ready),
      .R_audio_in_data(R_audio_out_data), .R_buffer_out(R_buffer_out),
.R_buffer_ready(R_buffer_ready));

      // Network Transmitter
nwk_tx network_tx(.clock(clock_65mhz), .reset(reset),
```

```
       .R_buffer_ready(R_buffer_ready),
       .R_buffer_in(R_buffer_out),
       .Gtx_buffer0_in(GTX_buffer0),
       .Gtx_buffer1_in(GTX_buffer1),
       .Gtx_buffer2_in({4'b0,GTX_buffer2}),
       .Gtx_buffer3_in({3'b0,GTX_buffer3}),
       .Gtx_buffer4_in(GTX_buffer4),
       .vsync(vsync), .fsm_state(GTX_buffer2),
       .tx_ready(tx_ready), .symbol_tx(symbol_tx), .tx_request(tx_request));

       // Serializer
serializer ser(.clock(clock_65mhz), .reset(reset), .tx_request(tx_request),
       .symbol_tx(symbol_tx), .rs232_txd(line), .tx_ready(tx_ready));

assign rs232_txd = line;
assign rs232_rts = tx_ready;

       // De-Serializer
deserializer deser(.clock(clock_65mhz), .reset(reset), .rs232_rxd(rs232_rxd),
       .symbol_rx(symbol_rx), .rx_data_ready(rx_data_ready),
.rx_ready(rs232_cts));

       // Network Receiver
nwk_rx network_rx(.clock(clock_65mhz), .reset(reset),
       .P_buffer_ready(P_buffer_ready),
       .P_buffer_out(P_buffer_out),
       .Grx_buffer0_out(GRX_buffer0),
       .Grx_buffer1_out(GRX_buffer1),
       .Grx_buffer2_out(GRX_buffer2),
       .Grx_buffer3_out(GRX_buffer3),
       .Grx_buffer4_out(GRX_buffer4),
       .rx_data_ready(rx_data_ready), .symbol_rx(symbol_rx));

       // Player Buffer
play_buffer audio_play_buffer(.clock(clock_65mhz), .reset(reset),
.P_sample_ready(P_sample_ready),
       .P_audio_out_data(P_audio_out_data), .P_buffer_in(P_buffer_out),
.P_buffer_ready(P_buffer_ready));

       // Playback Module
player audio_play(.clock(clock_65mhz), .reset(reset),
.audio_ready(audio_ready),
       .to_ac97_data(to_ac97_data), .P_audio_in_data(P_audio_out_data),
.P_sample_ready(P_sample_ready));

endmodule
```

B. Audio Controller

Audio Recorder

```
module recorder(
  input wire clock, // 65mhz system clock
  input wire reset, // 1 to reset to initial state
  input wire audio_ready, // high when AC97 data is available
  input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
  output wire [7:0] R_audio_out_data, // 8-bit PCM data output
  output reg R_sample_ready // filtered sample ready
);
        reg [2:0] sample_counter; //counts incoming samples

        //low-pass filter module
        reg [7:0] filter_input;
        wire [17:0] filter_output;
        fir31 filter(.clock(clock), .reset(reset), .audio_ready(audio_ready),
                .x(filter_input), .y(filter_output));

        //output audio sample
        assign R_audio_out_data = filter_output[17:10];

        always @ (posedge clock) begin
                //record every 8th sample
                sample_counter <= audio_ready ? sample_counter + 1 : sample_counter;

                if (reset) begin
                        R_sample_ready <= 0;
                        sample_counter <= 0;
                        filter_input <= 0;
                end else begin
                        if (audio_ready && (sample_counter == 7) ) begin
                                sample_counter <= 0; //reset counter
                                R_sample_ready <= 1; //signal sample is ready
                                filter_input <= from_ac97_data; //pass raw sample to filter
                        end else if (audio_ready)
                                filter_input <= from_ac97_data; //pass raw sample to filter
                        else begin
                                filter_input <= filter_input; //hold filter input
                                R_sample_ready <= 0; //no new sample
                        end
                end
        end
    endmodule
```

Record Buffer

```
//default buffer size 8 x 8-bit registers
module record_buffer #(parameter R_BUFFER_SIZE = 8) (
        input wire clock, // 65mhz system clock
        input wire reset, // 1 to reset to initial state
        input wire R_sample_ready,
        input wire [7:0] R_audio_in_data,
        output reg [(8*R_BUFFER_SIZE)-1:0] R_buffer_out,
        output reg R_buffer_ready
);
        //***R_BUFFER_SIZE dependent***\\
        reg [2:0] index;
        reg [5:0] i = 0;
        reg [7:0] buffer[R_BUFFER_SIZE - 1:0];

        always @(posedge clock) begin
                if (reset) begin
                        R_buffer_out <= 0;
                        R_buffer_ready <= 0;
                        index <= 0;
                        for (i = 0; i < R_BUFFER_SIZE; i=i+1)
```

```
                    buffer[i] <= 0;
            end else begin
                if (R_sample_ready) begin //trigger on "sample ready" signal
                    if (index == (R_BUFFER_SIZE-1)) begin
                        R_buffer_ready <= 1; //one-clock pulse

                        //***R_BUFFER_SIZE dependent***\\
                        //latch contents of buffer to output
                        R_buffer_out[7:0] <= buffer[0];
                        R_buffer_out[15:8] <= buffer[1];
                        R_buffer_out[23:16] <= buffer[2];
                        R_buffer_out[31:24] <= buffer[3];
                        R_buffer_out[39:32] <= buffer[4];
                        R_buffer_out[47:40] <= buffer[5];
                        R_buffer_out[55:48] <= buffer[6];
                        R_buffer_out[(8*R_BUFFER_SIZE)-1:(8*R_BUFFER_SIZE)-8] <=
R_audio_in_data;

                        buffer[R_BUFFER_SIZE-1] <= R_audio_in_data;

                        index <= 0; //reset index of buffer
                    end else begin
                        buffer[index] <= R_audio_in_data; //latch audio sample to
buffer
                        index <= index + 1; //point index to next available slot
                    end
                end else begin
                    R_buffer_ready <= 0;
                end
            end
        end
    endmodule
```

Play Buffer

```
//default buffer size 8 x 8-bit registers
module play_buffer #(parameter P_BUFFER_SIZE = 8) (
        input wire clock,
        input wire reset,
        output wire P_sample_ready,
        output wire [7:0] P_audio_out_data,
        input wire [(8*P_BUFFER_SIZE)-1:0] P_buffer_in,
        input wire P_buffer_ready,
);
        //***P_BUFFER_SIZE dependent***\\
        reg [2:0] index, i;
        reg [7:0] buffer[P_BUFFER_SIZE - 1:0];
        reg [13:0] counter;
        wire tick_6khz;

        assign tick_6khz = (counter == 10799); //6kHz one-clock pulse

        assign P_sample_ready = tick_6khz; //6kHz pulse is ready signal for sample playback
        assign P_audio_out_data = buffer[index]; //output buffer data at index

        always @(posedge clock) begin

            if (reset) begin
                counter <= 0;
                index <= 0;
                for (i = 0; i < P_BUFFER_SIZE; i=i+1)
                    buffer[i] <= 0;
            end else begin
                counter <= (counter < 10800) ? (counter + 1) : 0; //6kHz count @ 64.8MHz
clock
                if (P_buffer_ready) begin //trigger on incoming buffer ready
                    //***P_BUFFER_SIZE dependent***\\
                    //latch audio samples to buffer
                    buffer[0] <= P_buffer_in[7:0];
                    buffer[1] <= P_buffer_in[15:8];
```

```
                            buffer[2] <= P_buffer_in[23:16];
                            buffer[3] <= P_buffer_in[31:24];
                            buffer[4] <= P_buffer_in[39:32];
                            buffer[5] <= P_buffer_in[47:40];
                            buffer[6] <= P_buffer_in[55:48];
                            buffer[7] <= P_buffer_in[63:56];

                            //reset playback counter and buffer index
                            index <= 0;
                            counter <= 0;
                    end else begin
                            if ((index == (P_BUFFER_SIZE-1)) & tick_6khz) begin
                                    //hold last audio sample at output while new packet arrives
                                    index <= P_BUFFER_SIZE-1;
                            end else
                                    //increase index at a rate of 6kHz (for playback)
                                    index <= tick_6khz ? (index + 1) : index;
                    end
            end
        end
    endmodule
```

Audio Player

```
module player(
  input wire clock,    // 65mhz system clock
  input wire reset, // 1 to reset to initial state
  input wire audio_ready, // 1 when AC97 data is available
  output reg [7:0] to_ac97_data, // 8-bit PCM data to headphone
  input wire [7:0] P_audio_in_data,   // 8-bit PCM data input
  input wire P_sample_ready   // filtered sample ready
);
        reg [2:0] sample_counter;

        // low-pass reconstruction filter
        reg [7:0] filter_input;
        wire [17:0] filter_output;
        fir31 filter(.clock(clock), .reset(reset), .audio_ready(audio_ready),
                .x(filter_input), .y(filter_output));

        wire [7:0] filter_out; //final filtered sample
        assign filter_out = filter_output[14:7];

        always @ (posedge clock) begin
                //count samples received
                sample_counter <= audio_ready ? sample_counter + 1 : sample_counter;
                //trigger on AC97 ready signal
                if (audio_ready && (sample_counter == 7) ) begin
                        //latch incoming audio sample every 8th trigger of AC97 ready signal
                        sample_counter <= 0;
                        filter_input <= P_audio_in_data;
                end else if (audio_ready) begin
                        //input zero-expanded samples to filter
                        filter_input <= 0;
                        to_ac97_data <= filter_out;
                end
        end
    endmodule
```

C. Network Controller

Network Transmitter

```
module nwk_tx(
        input wire clock, // 65mhz system clock
        input wire reset, // 1 to reset to initial state
```

```
        input wire R_buffer_ready, // Audio Buffer Ready
        input wire [63:0] R_buffer_in, // Audio Buffer
        input wire [127:0] Gtx_buffer0_in, // Game Data 0 Buffer
        input wire [127:0] Gtx_buffer1_in, // Game Data 1 Buffer
        input wire [7:0] Gtx_buffer2_in, // Game Data 2 Buffer
        input wire [7:0] Gtx_buffer3_in, // Game Data 3 Buffer
        input wire [79:0] Gtx_buffer4_in, // Game Data 4 Buffer

        input wire [3:0] fsm_state, // game state
        input wire vsync, // frame sync signal

        input wire tx_ready, // Ready To Receive data signal (buffer empty)
        output reg [7:0] symbol_tx, // Symbol to TX
        output wire tx_request // Request to TX
);

        localparam IDL = 0;    //idle state
        localparam RTX = 1;    //transmit [recorded] audio packet; RTX=Recording Transmit
        localparam GTX0 = 2; //transmit game packets buffer0; GTX0=Game0 Transmit
        localparam GTX1 = 3; //transmit game packets buffer1; GTX1=Game1 Transmit
        localparam GTX2 = 4; //transmit game packets buffer2; GTX2=Game2 Transmit
        localparam GTX3 = 5; //transmit game packets buffer3; GTX3=Game3 Transmit
        localparam GTX4 = 6; //transmit rfid packet; GTX4=Game4 Transmit

        reg tx_ready_prev, vsync_prev; // previous state of signals
        wire done; // Packet TX Complete signal

        // arrays of state registers; [0]-current state, [1]-previous
        reg [2:0] state[1:0]; // network state
        reg [3:0] state_rtx_byte[1:0]; // audio byte TX
        reg [4:0] state_gtx_byte[1:0], Gtx_buffer_ready; // game byte TX

        // maximum-sized TX buffer
        reg [127:0] tx_buffer;

        //ModelSim debug code
        initial begin
       symbol_tx = 8'bX;
       state[1] = 0;
       state[0] = 0;
       state_rtx_byte[1] = 0;
       state_rtx_byte[0] = 0;
       state_gtx_byte[1] = 0;
       state_gtx_byte[0] = 0;
       tx_buffer = 0;
       Gtx_buffer_ready = 0;
       vsync_prev = 0;
    end

    // TX complete signal; one-clock pulse
    assign done =
       (state_rtx_byte[0] == 4'b0001) ||
       (state_gtx_byte[0] == 5'b00001);

        // send TX request signal to Serializer
        assign tx_request =
           ({state_rtx_byte[1],state_rtx_byte[0]} == {4'b0000,4'b0111}) ||
           ((state_rtx_byte[1]+1) == state_rtx_byte[0]) ||
           ({state_gtx_byte[1],state_gtx_byte[0]} == {5'b00000,5'b01111}) ||
           ((state_gtx_byte[1]+1) == state_gtx_byte[0]);

        always @(posedge clock) begin
               if (reset) begin
                      state[0] <= 0;
                      state_gtx_byte[0] <= 0;
                      state_rtx_byte[0] <= 0;
                      Gtx_buffer_ready <= 0;
                      vsync_prev <= 0;
               end else begin
                  // store previous states
                      state[1] <= state[0]; //state[1] is previous state
```

```
                        state_gtx_byte[1] <= state_gtx_byte[0];
                        state_rtx_byte[1] <= state_rtx_byte[0];
                        tx_ready_prev <= tx_ready;
                        vsync_prev <= vsync;

                        // deactivate game packet TX request signals after
                        // successfully transmitting ALL GAME packets
                        if ( done && (
                          ((state[0] == GTX0) && (~|Gtx_buffer_ready[4:1]))||
                          ((state[0] == GTX1) && (~|Gtx_buffer_ready[4:2]))||
                          ((state[0] == GTX2) && (~|Gtx_buffer_ready[4:3]))||
                          ((state[0] == GTX3) && (~|Gtx_buffer_ready[4]))||
                              (state[0] == GTX4) ) )
                            Gtx_buffer_ready <= 4'b0;
                        else begin
                          // packet mapping:
Gtx_buffer_ready={buffer4,buffer3,buffer2,buffer1,buffer0}
                          // a combination of GAME packets are sent according to state of the
game.
                          // packet send request signals are "mapped" to state.
                          // game packets sent on positive edge of vsync.
                          case (fsm_state)
                            0:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            1:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            2:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            3:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            4:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            5:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b10100 :
Gtx_buffer_ready;
                            6:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            7:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00111 :
Gtx_buffer_ready;
                            8:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b01100 :
Gtx_buffer_ready;
                            9:      Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            10:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            11:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            12:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            13:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            14:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            15:     Gtx_buffer_ready <= &{~vsync_prev,vsync} ? 5'b00100 :
Gtx_buffer_ready;
                            default: Gtx_buffer_ready <= 0;
            endcase
          end

                        // handle network states.
                        // game packet send requests are prioritized.
                        case (state[0])
                             IDL:  state[0] <= Gtx_buffer_ready[0] ? GTX0 :
                                 ( Gtx_buffer_ready[1] ? GTX1 :
                                 ( Gtx_buffer_ready[2] ? GTX2 :
                                 ( Gtx_buffer_ready[3] ? GTX3 :
                                 ( Gtx_buffer_ready[4] ? GTX4 :
                                 ( R_buffer_ready ? RTX : IDL)))));
                             RTX:  state[0] <= done ? IDL : RTX;
                             GTX0: state[0] <= done ?
                                       (Gtx_buffer_ready[1] ? GTX1 :
                                       (Gtx_buffer_ready[2] ? GTX2 :
```

```
                                        (Gtx_buffer_ready[3] ? GTX3 :
                                        (Gtx_buffer_ready[4] ? GTX4 : IDL)))) : GTX0;
                                GTX1: state[0] <= done ?
                                        (Gtx_buffer_ready[2] ? GTX2 :
                                        (Gtx_buffer_ready[3] ? GTX3 :
                                        (Gtx_buffer_ready[4] ? GTX4 : IDL))) : GTX1;
                                GTX2: state[0] <= done ?
                                        (Gtx_buffer_ready[3] ? GTX3 :
                                        (Gtx_buffer_ready[4] ? GTX4 : IDL)) : GTX2;
                                GTX3: state[0] <= done ?
                                        (Gtx_buffer_ready[4] ? GTX4 : IDL) : GTX3;
                                GTX4: state[0] <= done ? IDL : GTX4;
                                default: state[0] <= 0;
                        endcase

                        // audio packet transmit
                        if (state[0] == RTX) begin
                                //load audio data to TX buffer
                                tx_buffer[63:0] <= (state[1] == IDL) ? R_buffer_in :
                                   (( (state_rtx_byte[0] > 4'b1000) && tx_request ) ? (tx_buffer >>
8) : tx_buffer);

                                //load symbol buffer with header
                                if (state_rtx_byte[0] == 4'b0111)
                                        symbol_tx <= 8'b0000_0001;    //header byte
                                //load symbol buffer with TX buffer data
                                else if (state_rtx_byte[0] > 4'b0111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                //handle byte-send states
                           case (state_rtx_byte[0])
                                        4'b0000: state_rtx_byte[0] <= tx_ready ? 4'b0111 :
state_rtx_byte[0];     //waiting
                                        4'b0111: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send HEADER
                                        4'b1000: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE0
                                        4'b1001: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE1
                                        4'b1010: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE2
                                        4'b1011: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE3
                                        4'b1100: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE4
                                        4'b1101: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE5
                                        4'b1110: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_rtx_byte[0]+1) : state_rtx_byte[0]; //send BYTE6
                                        4'b1111: state_rtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 4'b0001 : state_rtx_byte[0];         //send BYTE7
                                        4'b0001: state_rtx_byte[0] <= 4'b0000; // TX done state
                                        default: state_rtx_byte[0] <= 0;
                                endcase

                        end else if (state[0] == GTX0) begin
                                tx_buffer[127:0] <= (state[1] != GTX0) ? Gtx_buffer0_in :
                                        (( (state_gtx_byte[0] > 5'b10000) && tx_request ) ?
(tx_buffer >> 8) : tx_buffer);

                                if (state_gtx_byte[0] == 5'b01111)
                                        symbol_tx <= 8'b0000_1000;    //header byte
                                else if (state_gtx_byte[0] > 5'b01111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                           case (state_gtx_byte[0])
```

```
                                        5'b00000: state_gtx_byte[0] <= tx_ready ? 5'b01111 :
state_gtx_byte[0];     //waiting
                                        5'b01111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send HEADER
                                        5'b10000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE0
                                        5'b10001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE1
                                        5'b10010: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE2
                                        5'b10011: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE3
                                        5'b10100: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE4
                                        5'b10101: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE5
                                        5'b10110: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE6
                                        5'b10111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE7
                                        5'b11000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE8
                                        5'b11001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE9
                                        5'b11010: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE10
                                        5'b11011: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE11
                                        5'b11100: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE12
                                        5'b11101: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE13
                                        5'b11110: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE14
                                        5'b11111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 5'b00001 : state_gtx_byte[0];       //send BYTE15
                                        5'b00001: state_gtx_byte[0] <= 5'b00000;     //end
                                        default: state_gtx_byte[0] <= 0;
                                endcase
                        end else if (state[0] == GTX1) begin
                                tx_buffer[127:0] <= (state[1] != GTX1) ? Gtx_buffer1_in :
                                        (( (state_gtx_byte[0] > 5'b10000) && tx_request ) ?
(tx_buffer >> 8) : tx_buffer);

                                if (state_gtx_byte[0] == 5'b01111)
                                        symbol_tx <= 8'b0000_1001;     //header byte
                                else if (state_gtx_byte[0] > 5'b01111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                            case (state_gtx_byte[0])
                                        5'b00000: state_gtx_byte[0] <= tx_ready ? 5'b01111 :
state_gtx_byte[0];     //waiting
                                        5'b01111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send HEADER
                                        5'b10000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE0
                                        5'b10001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE1
                                        5'b10010: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE2
                                        5'b10011: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE3
                                        5'b10100: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE4
                                        5'b10101: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE5
                                        5'b10110: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE6
```

```
                                        5'b10111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE7
                                        5'b11000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE8
                                        5'b11001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE9
                                        5'b11010: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE10
                                        5'b11011: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE11
                                        5'b11100: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE12
                                        5'b11101: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE13
                                        5'b11110: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE14
                                        5'b11111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 5'b00001 : state_gtx_byte[0];         //send BYTE15
                                        5'b00001: state_gtx_byte[0] <= 5'b00000;     //end
                                        default: state_gtx_byte[0] <= 0;
                                endcase
                        end else if (state[0] == GTX2) begin
                                tx_buffer[7:0] <= (state[1] != GTX2) ? Gtx_buffer2_in :
                                        (( (state_gtx_byte[0] > 5'b10000) && tx_request ) ?
(tx_buffer >> 8) : tx_buffer);

                                if (state_gtx_byte[0] == 5'b01111)
                                        symbol_tx <= 8'b0000_1010;     //header byte
                                else if (state_gtx_byte[0] > 5'b01111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                            case (state_gtx_byte[0])
                                        5'b00000: state_gtx_byte[0] <= tx_ready ? 5'b01111 :
state_gtx_byte[0];     //waiting
                                        5'b01111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send HEADER
                                        5'b10000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 5'b00001 : state_gtx_byte[0];         //send BYTE0
                                        5'b00001: state_gtx_byte[0] <= 5'b00000;     //end
                                        default: state_gtx_byte[0] <= 0;
                                endcase
                        end else if (state[0] == GTX3) begin
                                tx_buffer[7:0] <= (state[1] != GTX3) ? Gtx_buffer3_in :
                                        (( (state_gtx_byte[0] > 5'b10000) && tx_request ) ?
(tx_buffer >> 8) : tx_buffer);

                                if (state_gtx_byte[0] == 5'b01111)
                                        symbol_tx <= 8'b0000_1011;     //header byte
                                else if (state_gtx_byte[0] > 5'b01111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                            case (state_gtx_byte[0])
                                        5'b00000: state_gtx_byte[0] <= tx_ready ? 5'b01111 :
state_gtx_byte[0];     //waiting
                                        5'b01111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send HEADER
                                        5'b10000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 5'b00001 : state_gtx_byte[0];         //send BYTE0
                                        5'b00001: state_gtx_byte[0] <= 5'b00000;     //end
                                        default: state_gtx_byte[0] <= 0;
                                endcase
                        end else if (state[0] == GTX4) begin
                                tx_buffer[79:0] <= (state[1] != GTX4) ? Gtx_buffer4_in :
                                        (( (state_gtx_byte[0] > 5'b10000) && tx_request ) ?
(tx_buffer >> 8) : tx_buffer);

                                if (state_gtx_byte[0] == 5'b01111)
```

```
                                        symbol_tx <= 8'b0000_1100;    //header byte
                                else if (state_gtx_byte[0] > 5'b01111)
                                        symbol_tx <= tx_buffer[7:0];
                                else
                                        symbol_tx <= 8'bX;

                                case (state_gtx_byte[0])
                                        5'b00000: state_gtx_byte[0] <= tx_ready ? 5'b01111 :
state_gtx_byte[0];     //waiting
                                        5'b01111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send HEADER
                                        5'b10000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE0
                                        5'b10001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE1
                                        5'b10010: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE2
                                        5'b10011: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE3
                                        5'b10100: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE4
                                        5'b10101: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE5
                                        5'b10110: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE6
                                        5'b10111: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE7
                                        5'b11000: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? (state_gtx_byte[0]+1) : state_gtx_byte[0]; //send BYTE8
                                        5'b11001: state_gtx_byte[0] <= (~tx_ready_prev && tx_ready)
? 5'b00001 : state_gtx_byte[0];       //send BYTE9
                                        5'b00001: state_gtx_byte[0] <= 5'b00000;     //end
                                        default: state_gtx_byte[0] <= 0;
                                endcase
                        end

                end
        end

    endmodule
```

Serializer

```
module serializer #(parameter CLOCK_COUNT = 288)(
        input wire clock,
        input wire reset,
        input wire tx_request,
        input wire [7:0] symbol_tx,

        output wire rs232_txd,
        output wire tx_ready   //rs232_rts
);
   reg [21:0] clock_counter;
        wire tick_DC; //data clock
        assign tick_DC = (clock_counter == (CLOCK_COUNT-1)); //data clock tick

        reg muxbit;
        reg [3:0] state;
        assign tx_ready = (state == 0);

        //ModelSim debug code
        initial state = 0;

        always @(posedge clock) begin
                if (reset) begin
                        clock_counter <= 0;
                        state <= 0;
                end else begin
                        //data clock counter
```

```
                        clock_counter <= tx_request ? 0 : ( (clock_counter < CLOCK_COUNT) ?
(clock_counter + 1) : 0 );
                        //tx bits state machine
                        case(state)
                                4'b0000: state <= tx_request  ? 4'b0100 : state;
                                4'b0100: state <= tick_DC ? 4'b1000 : state; // start
                                4'b1000: state <= tick_DC ? 4'b1001 : state; // bit 0
                                4'b1001: state <= tick_DC ? 4'b1010 : state; // bit 1
                                4'b1010: state <= tick_DC ? 4'b1011 : state; // bit 2
                                4'b1011: state <= tick_DC ? 4'b1100 : state; // bit 3
                                4'b1100: state <= tick_DC ? 4'b1101 : state; // bit 4
                                4'b1101: state <= tick_DC ? 4'b1110 : state; // bit 5
                                4'b1110: state <= tick_DC ? 4'b1111 : state; // bit 6
                                4'b1111: state <= tick_DC ? 4'b0001 : state; // bit 7
                                4'b0001: state <= tick_DC ? 4'b0000 : state; // stop1
                                default: state <= tick_DC ? 4'b0000 : state;
                        endcase
                end
        end

        always @*
                case(state[2:0])
                        0: muxbit <= symbol_tx[0];
                        1: muxbit <= symbol_tx[1];
                        2: muxbit <= symbol_tx[2];
                        3: muxbit <= symbol_tx[3];
                        4: muxbit <= symbol_tx[4];
                        5: muxbit <= symbol_tx[5];
                        6: muxbit <= symbol_tx[6];
                        7: muxbit <= symbol_tx[7];
                endcase

        // combine start, data, and stop bits together
        assign rs232_txd = (state < 4) | (state[3] & muxbit);

     endmodule
```

De-Serializer

```
module deserializer #(parameter CLOCK_COUNT = 36)(
        input clock, reset,
        input wire rs232_rxd,
        input wire rx_ready, //rs232_cts

        output reg [7:0] symbol_rx,
        output reg rx_data_ready
);

   wire tick_OS; //oversample clock

        reg [21:0] clock_counter; //oversample clock counter
        assign tick_OS = (clock_counter == (CLOCK_COUNT-1));

        reg [2:0] bit_spacing;
        wire next_bit = (bit_spacing == 7); //oversample data clock x8

        //rxd line sync registers
        reg [1:0] RxD_sync, RxD_cnt;
        reg RxD_bit;
        reg [3:0] state;

        //ModelSim debug code
        initial begin
            RxD_bit = 1;
            RxD_sync = 2'b11;
            RxD_cnt = 2'b11;
            state = 0;
            bit_spacing = 0;
            clock_counter = 0;
```

```
            symbol_rx = 8'b00000000;
            rx_data_ready = 0;
        end

        always @(posedge clock) begin
                if (reset) begin
                        symbol_rx <= 8'b0;
                        rx_data_ready <= 8'b0;
                        clock_counter <= 0;
                        RxD_bit <= 1;
                end else begin
                        //oversample clock counter
                        clock_counter <= (clock_counter < CLOCK_COUNT) ? (clock_counter + 1) : 0;
                        //synchronize rxd line
                        RxD_sync <= tick_OS ? {RxD_sync[0], rs232_rxd} : RxD_sync;

                        if (tick_OS) begin
                                //oversample rxd line, filter noise
                                if (RxD_sync[1] && RxD_cnt!=2'b11)
                                        RxD_cnt <= RxD_cnt + 1;
                                else if (~RxD_sync[1] && RxD_cnt!=2'b00)
                                        RxD_cnt <= RxD_cnt - 1;

                                if (RxD_cnt == 2'b00)
                                        RxD_bit <= 0;
                                else if (RxD_cnt == 2'b11)
                                        RxD_bit <= 1;

                                //incoming bits state machine
                                case(state)
                                        4'b0000: if(~RxD_bit) state <= 4'b1000; // start bit found
                                        4'b1000: if(next_bit) state <= 4'b1001; // bit 0
                                        4'b1001: if(next_bit) state <= 4'b1010; // bit 1
                                        4'b1010: if(next_bit) state <= 4'b1011; // bit 2
                                        4'b1011: if(next_bit) state <= 4'b1100; // bit 3
                                        4'b1100: if(next_bit) state <= 4'b1101; // bit 4
                                        4'b1101: if(next_bit) state <= 4'b1110; // bit 5
                                        4'b1110: if(next_bit) state <= 4'b1111; // bit 6
                                        4'b1111: if(next_bit) state <= 4'b0001; // bit 7
                                        4'b0001: if(next_bit) state <= 4'b0000; // stop bit
                                        default: state <= 4'b0000;
                                endcase
                                //shift register incoming bits
                                if (next_bit && state[3])
                                        symbol_rx <= {RxD_bit, symbol_rx[7:1]};
                        end
                        //latch bits at appropriate data clock
                        bit_spacing <= (state==0) ? 0 : ( tick_OS ? (bit_spacing + 1) :
bit_spacing );
                        // ready only if the stop bit is received
                        rx_data_ready <= (tick_OS && next_bit && (state == 4'b0001) && RxD_bit);
                end
        end
     endmodule
```

Network Receiver

```
module nwk_rx(
        input wire clock, // 65mhz system clock
        input wire reset, // 1 to reset to initial state

        output wire P_buffer_ready, // Audio Buffer Ready
        output reg [63:0] P_buffer_out, //Playback audio buffer
        output reg [127:0] Grx_buffer0_out, // Game Data 0 Buffer
        output reg [127:0] Grx_buffer1_out, // Game Data 1 Buffer
        output reg [7:0] Grx_buffer2_out, // Game Data 2 Buffer
        output reg [7:0] Grx_buffer3_out, // Game Data 3 Buffer
        output reg [79:0] Grx_buffer4_out, // Game Data 4 Buffer
```

```
        input wire rx_data_ready, // Data Available signal (buffer full)
        input wire [7:0] symbol_rx // Symbol to RX
);

        localparam IDL  = 5'b00000; //idle state
        localparam RX   = 5'b00001; //receive mode (read header)
        localparam PRX  = 5'b10000; //receive audio packet
        localparam GRX0 = 5'b10001; //receive game packet of type 0 (size 128)
        localparam GRX1 = 5'b10010; //receive game packet of type 1 (size 64)
        localparam GRX2 = 5'b10011; //receive game packet of type 2 (size 80)

        wire done; //RX Complete signal
        // arrays of state registers; [0]-current state, [1]-previous
        reg [4:0] state[1:0]; //network state
        reg [4:0] state_prx_byte[1:0]; //byte RX state

        reg [127:0] rx_buffer; //maximum sized RX buffer
        wire [7:0] header; // header buffer

        //pulse RX Complete signal after last byte RX'd
        assign done =
           ((header == 8'b0001)&&(state_prx_byte[0] == 5'b01000)) ||
           ((header == 8'b1000)&&(state_prx_byte[0] == 5'b10000)) ||
           ((header == 8'b1001)&&(state_prx_byte[0] == 5'b10000)) ||
           ((header == 8'b1010)&&(state_prx_byte[0] == 5'b00001)) ||
           ((header == 8'b1011)&&(state_prx_byte[0] == 5'b00001)) ||
               ((header == 8'b1100)&&(state_prx_byte[0] == 5'b01010));

        //buffer header byte in RX mode
        assign header = (state[0] == RX) ? symbol_rx : header;
        //pulse Audio Packet RX signal when ready
        assign P_buffer_ready = (state_prx_byte[1] == 5'b01000);

        //ModelSim debug code
        initial begin
            state[1] <= IDL;
                state[0] <= IDL;
                state_prx_byte[1] <= 5'b0;
                state_prx_byte[0] <= 5'b0;
                rx_buffer <= 128'b0;
                P_buffer_out <= 64'b0;
                Grx_buffer0_out <= 128'b0;
                Grx_buffer1_out <= 128'b0;
                Grx_buffer2_out <= 8'b0;
                Grx_buffer3_out <= 8'b0;
                Grx_buffer4_out <= 80'b0;
   end

        always @(posedge clock) begin
               if (reset) begin
                   state[1] <= IDL;
                   state[0] <= IDL;
                   state_prx_byte[1] <= 5'b0;
                   state_prx_byte[0] <= 5'b0;
                   rx_buffer <= 128'b0;
                   P_buffer_out <= 64'b0;
    Grx_buffer0_out <= 128'b0;
    Grx_buffer1_out <= 128'b0;
    Grx_buffer2_out <= 8'b0;
    Grx_buffer3_out <= 8'b0;
                       Grx_buffer4_out <= 80'b0;
               end else begin
                   state[1] <= state[0]; //store previous states
                       state_prx_byte[1] <= state_prx_byte[0];

                   //handle network RX states; header controls flow of state
                   case (state[0])
                      IDL : state[0] <= rx_data_ready ? RX : IDL;
                               RX  : state[0] <= (header == 8'b0001) ? PRX :
                                  ((header == 8'b1000 || header == 8'b1001) ? GRX0 :
                                  ((header == 8'b1010 || header == 8'b1011) ? GRX1 :
```

```
                    ((header == 8'b1100) ? GRX2 : RX)));
                PRX : state[0] <= done ? IDL : PRX;
                GRX0: state[0] <= done ? IDL : GRX0;
                GRX1: state[0] <= done ? IDL : GRX1;
                GRX2: state[0] <= done ? IDL : GRX2;
                default: state[0] <= 0;
        endcase

        //latch buffers to output
        P_buffer_out <= ( (header == 8'b0001)&&(state_prx_byte[0] == 5'b01000) )
?
           rx_buffer[63:0] : P_buffer_out;

        Grx_buffer0_out <= ( (header == 8'b1000)&&(state_prx_byte[0] == 5'b10000)
) ?
           rx_buffer[127:0] : Grx_buffer0_out;

        Grx_buffer1_out <= ( (header == 8'b1001)&&(state_prx_byte[0] == 5'b10000)
) ?
           rx_buffer[127:0] : Grx_buffer1_out;

        Grx_buffer2_out <= ( (header == 8'b1010)&&(state_prx_byte[0] == 5'b00001)
) ?
           rx_buffer[7:0] : Grx_buffer2_out;

        Grx_buffer3_out <= ( (header == 8'b1011)&&(state_prx_byte[0] == 5'b00001)
) ?
           rx_buffer[7:0] : Grx_buffer3_out;

        Grx_buffer4_out <= ( (header == 8'b1100)&&(state_prx_byte[0] == 5'b01010)
) ?
           rx_buffer[79:0] : Grx_buffer4_out;

        if (state[0] == PRX) begin //RX audio packet
           if (rx_data_ready) begin //buffer incoming symbol when ready
              rx_buffer <= rx_buffer >> 8;
              rx_buffer[63:56] <= symbol_rx;
           end else
              rx_buffer <= rx_buffer;

            //byte receive state machine
           case (state_prx_byte[0])
                      5'b00000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE0
                 5'b00001: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE1
                 5'b00010: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE2
                 5'b00011: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE3
                 5'b00100: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE4
                 5'b00101: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE5
                 5'b00110: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE6
                 5'b00111: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE7
                 default: state_prx_byte[0] <= 5'b00000; //end
                endcase
        end else if (state[0] == GRX0) begin
           if (rx_data_ready) begin
              rx_buffer <= rx_buffer >> 8;
              rx_buffer[127:120] <= symbol_rx;
           end else
              rx_buffer <= rx_buffer;

           case (state_prx_byte[0])
                      5'b00000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE0
```

```
                                   5'b00001: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE1
                                   5'b00010: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE2
                                   5'b00011: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE3
                                   5'b00100: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE4
                                   5'b00101: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE5
                                   5'b00110: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE6
                                   5'b00111: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE7
                                   5'b01000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE8
                                   5'b01001: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE9
                                   5'b01010: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE10
                                   5'b01011: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE11
                                   5'b01100: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE12
                                   5'b01101: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE13
                                   5'b01110: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE14
                                   5'b01111: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE15
                                   default: state_prx_byte[0] <= 5'b00000; //end
                                 endcase
                         end else if (state[0] == GRX1) begin
                            if (rx_data_ready) begin
                               rx_buffer <= rx_buffer >> 8;
                               rx_buffer[7:0] <= symbol_rx;
                            end else
                               rx_buffer <= rx_buffer;

                            case (state_prx_byte[0])
                                        5'b00000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE0
                                   default: state_prx_byte[0] <= 5'b00000; //end
                                 endcase
                         end else if (state[0] == GRX2) begin
                            if (rx_data_ready) begin
                               rx_buffer <= rx_buffer >> 8;
                               rx_buffer[79:72] <= symbol_rx;
                            end else
                               rx_buffer <= rx_buffer;

                            case (state_prx_byte[0])
                                        5'b00000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE0
                                   5'b00001: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE1
                                   5'b00010: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE2
                                   5'b00011: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE3
                                   5'b00100: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE4
                                   5'b00101: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE5
                                   5'b00110: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE6
                                   5'b00111: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE7
                                   5'b01000: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE8
```

```
                                        5'b01001: state_prx_byte[0] <= rx_data_ready ?
(state_prx_byte[0]+1) : state_prx_byte[0];   //receive BYTE9
                                        default: state_prx_byte[0] <= 5'b00000; //end
                                      endcase
                              end

                    end
           end

      endmodule
```

D. Master FSM

```
// Javier Garcia , 6.111 Fall 2009

module fsm
(
    input clk,
    input reset,
    input enter,
    input one_player,
    input two_players,
    input game_over,
    input [3:0] ext_state,
        input [79:0] id,
        input win,
    output reg [3:0] state
);

  //states
  localparam START                   = 4'd0;
  localparam LOGIN                   = 4'd1;
  localparam ONE_PLAYER              = 4'd2;
  localparam ONE_PLAYER_GAME         = 4'd3;
  localparam TWO_PLAYERS             = 4'd4;
  localparam HOST_READY_WAITING      = 4'd5;
  localparam HOST_READY              = 4'd6;
  localparam TWO_PLAYER_GAME_HOST    = 4'd7;
  localparam TWO_PLAYER_GAME_CLIENT  = 4'd8;
  localparam GAME_OVER               = 4'd9;
  localparam YOU_WIN                 = 4'd10;


  reg [3:0] next_state;


  // sequential logic: state register
  always @ (posedge clk)
  begin
    if (reset)
      state <= START;
    else
      state <= next_state;
  end


  // combinational logic: next state
```

```
  always @ ( * )
  begin
    case(state)

       START:                                   next_state = ( id[1:0]==2'b01 ||
id[1:0]==2'b10 ) ? LOGIN

      : START;

             LOGIN:                       next_state = one_player ? ONE_PLAYER
                                                     : two_players ? TWO_PLAYERS
                                                     : LOGIN;

       ONE_PLAYER:                  next_state = enter ? ONE_PLAYER_GAME
                                                     : ONE_PLAYER;

       ONE_PLAYER_GAME:             next_state = game_over ? GAME_OVER

      : win ? YOU_WIN
                                                     : ONE_PLAYER_GAME;

       TWO_PLAYERS:                 next_state = enter ? HOST_READY_WAITING
                                                     : (ext_state == HOST_READY_WAITING)
? HOST_READY
                                                     : TWO_PLAYERS;

       HOST_READY_WAITING:          next_state = (ext_state ==
TWO_PLAYER_GAME_CLIENT)  ? TWO_PLAYER_GAME_HOST
                                                     : HOST_READY_WAITING;

       HOST_READY:                  next_state = enter ? TWO_PLAYER_GAME_CLIENT
                                                     : HOST_READY;

       TWO_PLAYER_GAME_HOST:        next_state = game_over ? GAME_OVER

      : win ? YOU_WIN
                                                     : TWO_PLAYER_GAME_HOST;

       TWO_PLAYER_GAME_CLIENT: next_state = (ext_state == GAME_OVER) ?
GAME_OVER

      : (ext_state == YOU_WIN) ? YOU_WIN
                                                     : TWO_PLAYER_GAME_CLIENT;

       GAME_OVER:                   next_state = enter ? START
                                                     : GAME_OVER;

             YOU_WIN:                   next_state = enter ? START
                                                     : YOU_WIN;

      default:                      next_state = START;

    endcase
  end
endmodule
```

E. Game Logic

```
module game_logic #(parameter DISPLAY_WIDTH = 1024,
                                                        DISPLAY_HEIGHT = 768,
                                                        P1_BULLET_RADIUS = 2,
                                                        P1_BULLET_COLOR =
3'b111,//DEFAULT: WHITE
                                                        PLAYER1_WIDTH = 36,
// default width: 32 pixels
                                                        PLAYER1_HEIGHT = 54,
// default height: 32 pixels
                                                        PLAYER1_COLOR =
3'b110,
                                                        PLAYER1_SPEED = 4,
                                                        P2_BULLET_RADIUS = 2,
                                                        P2_BULLET_COLOR =
3'b111,//DEFAULT: WHITE
                                                        PLAYER2_WIDTH = 36,
// default width: 32 pixels
                                                        PLAYER2_HEIGHT = 54,
// default height: 32 pixels
                                                        PLAYER2_COLOR =
3'b011,
                                                        PLAYER2_SPEED = 4,
                                                        E_BULLET_RADIUS = 2,
                                                        E_BULLET_COLOR =
3'b101,//DEFAULT: PINK
                                                        ENEMY_COLOR = 3'b100,
                                                        ENEMY_WIDTH = 64,
// default width: 32 pixels
                                                        ENEMY_HEIGHT = 64,
                                                        ENEMY_SPEED = 1,
                                    // default height: 32 pixels
                                                        BULLET_SPEED = 12,
                                                        ENEMY3_COLOR =
3'b101,
                                                        ENEMY3_WIDTH = 128,
// default width: 32 pixels
                                                        ENEMY3_HEIGHT = 64,
                                                        ENEMY3_SPEED = 2,
                                    // default height: 32 pixels
                                                        E3_BULLET_RADIUS = 4,
                                                        E3_BULLET_COLOR =
3'b100)
(
   input vclock,
    input reset,
    input button_0,
    input up,           // 1 when paddle should move up
    input down,    // 1 when paddle should move down
    input left,
    input right,
    input hsync,        // XVGA horizontal sync signal (active low)
    input vsync,        // XVGA vertical sync signal (active low)
    input blank,        // XVGA blanking (1 means output black pixel)
       input [127:0] game_packet1_in,
```

```
    input [127:0] game_packet2_in,
    input [4:0] inputs_packet_in,
    input [3:0] state_packet_in,
    output reg [127:0] game_packet1_out,
    output reg [127:0] game_packet2_out,
    output reg [4:0] inputs_packet_out,
    output reg game_over,
       output reg [4:0] p1_score,
       output reg [4:0] p2_score,
       output reg win
);

  localparam START                    = 4'd0;
  localparam LOGIN                    = 4'd1;
  localparam ONE_PLAYER               = 4'd2;
  localparam ONE_PLAYER_GAME          = 4'd3;
  localparam TWO_PLAYERS              = 4'd4;
  localparam HOST_READY_WAITING       = 4'd5;
  localparam HOST_READY               = 4'd6;
  localparam TWO_PLAYER_GAME_HOST     = 4'd7;
  localparam TWO_PLAYER_GAME_CLIENT = 4'd8;
  localparam GAME_OVER                = 4'd9;
  localparam YOU_WIN                                      = 4'd10;

reg ext_up;
reg ext_down;
reg ext_left;
reg ext_right;
reg ext_button_0;

reg t_vsync;
reg t_hsync;
reg t_blank;

always @ (posedge vclock) begin
            t_vsync <= vsync;
            t_hsync <=hsync;
            t_blank <= blank;
      end

//Player 1 coordinates and pixel info
      reg [9:0]  player1_y;
      reg [10:0] player1_x;
      wire [2:0] player1_pixel;
      reg player1_display;
//P1 bullet coordinates and pixel info
      reg [9:0]  p1_bullet_y;
      reg [10:0] p1_bullet_x;
      wire [2:0] p1_bullet_pixel;
      reg p1_bullet_display;

//Player 2 coordinates and pixel info
      reg [9:0]  player2_y;
      reg [10:0] player2_x;
      wire [2:0] player2_pixel;
      reg player2_display;
//P2 bullet coordinates and pixel info
```

```
	reg [9:0]  p2_bullet_y;
	reg [10:0] p2_bullet_x;
	wire [2:0] p2_bullet_pixel;
	reg p2_bullet_display;


//Enemy 1 coordinates and pixel info
	reg [9:0]  enemy1_y;
	reg [10:0] enemy1_x;
	wire [2:0] enemy1_pixel;
	reg enemy1_display;
//E1 bullet coordinates and pixel info
	reg [9:0]  e1_bullet_y;
	reg [10:0] e1_bullet_x;
	wire [2:0] e1_bullet_pixel;
	reg e1_bullet_display;



//Enemy 2 coordinates and pixel info
	reg [9:0]  enemy2_y;
	reg [10:0] enemy2_x;
	wire [2:0] enemy2_pixel;
	reg enemy2_display;
//E2 bullet coordinates and pixel info
	reg [9:0]  e2_bullet_y;
	reg [10:0] e2_bullet_x;
	wire [2:0] e2_bullet_pixel;
	reg e2_bullet_display;

//Enemy 3 coordinates and pixel info
	reg [9:0]  enemy3_y;
	reg [10:0] enemy3_x;
	wire [2:0] enemy3_pixel;
	reg enemy3_display;
//E3 bullet coordinates and pixel info
	reg [9:0]  e3_bullet_y;
	reg [10:0] e3_bullet_x;
	wire [2:0] e3_bullet_pixel;
	reg e3_bullet_display;



	reg [10:0] player1_x_limit;
	reg [9:0]  player1_y_limit;
	reg [10:0] player2_x_limit;
	reg [9:0]  player2_y_limit;
	reg [10:0] enemy1_x_limit;
	reg [9:0]  enemy1_y_limit;
	reg [10:0] enemy2_x_limit;
	reg [9:0]  enemy2_y_limit;
	reg [10:0] enemy3_x_limit;
	reg [9:0]  enemy3_y_limit;

	always @ ( * ) begin
		player1_x_limit = player1_x+PLAYER1_WIDTH;
```

```
            player1_y_limit = player1_y+PLAYER1_HEIGHT;
            player2_x_limit = player2_x+PLAYER2_WIDTH;
            player2_y_limit = player2_y+PLAYER2_HEIGHT;
            enemy1_x_limit = enemy1_x+ENEMY_WIDTH;
            enemy1_y_limit = enemy1_y+ENEMY_HEIGHT;
            enemy2_x_limit = enemy2_x+ENEMY_WIDTH;
            enemy2_y_limit = enemy2_y+ENEMY_HEIGHT;
            enemy3_x_limit = enemy3_x+ENEMY3_WIDTH;
            enemy3_y_limit = enemy3_y+ENEMY3_HEIGHT;
      end
      always @ ( * ) begin
            ext_button_0 = inputs_packet_in[0];
            ext_up = inputs_packet_in[1];
            ext_down = inputs_packet_in[2];
            ext_left = inputs_packet_in[3];
            ext_right = inputs_packet_in[4];
      end


      //timing wires
      reg [7:0] e12_respawn_counter;
      reg [7:0] e12_x_movement_counter;
      reg e12_x_movement;
      reg e3_x_movement;
      reg [7:0] e3_respawn_counter;
      reg [6:0] e3_x_movement_counter;


always @ (posedge vclock)
      begin
            if (reset || ( state_packet_in == GAME_OVER ) || (
state_packet_in == YOU_WIN ) )
            begin
                  //reset counters
                  //counter <= 0;
                  e12_respawn_counter <=0;
                  e12_x_movement_counter <= 0;
                  e12_x_movement <= 0;
                  player1_x <= DISPLAY_WIDTH/3 - PLAYER1_WIDTH/2 - 1;
//1024/3 -8 - 1;
                  player1_y <= DISPLAY_HEIGHT -PLAYER1_HEIGHT -1; //768/2 -32
-1;
                  player1_display <=1;
                  p1_bullet_x <= DISPLAY_WIDTH/3 - 1;
                  p1_bullet_y <= DISPLAY_HEIGHT+2;
                  p1_bullet_display <=0;
                  player2_x <= 2*DISPLAY_WIDTH/3 - PLAYER2_WIDTH/2 - 1;
//3*1024/4 -8 - 1;
                  player2_y <= DISPLAY_HEIGHT -PLAYER2_HEIGHT -1; //768/2 -32
-1;
                  player2_display <=1;
                  p2_bullet_x <= DISPLAY_WIDTH/3 - 1;
                  p2_bullet_y <= DISPLAY_HEIGHT+2;
                  p2_bullet_display <=0;
                  enemy1_x <= DISPLAY_WIDTH/4 - ENEMY_WIDTH/2 - 1; //1024/2 -
8 - 1;
                  enemy1_y <= 0; //768/2 -32 -1;
```

```
                enemy1_display <=1;
                e1_bullet_x <= DISPLAY_WIDTH/4 - 1;
                e1_bullet_y <= DISPLAY_HEIGHT+2;
                e1_bullet_display <=0;
                enemy2_x <= 3*DISPLAY_WIDTH/4 - ENEMY_WIDTH/2 - 1; //1024/2
-8 - 1;
                enemy2_y <= 0; //768/2 -32 -1;
                enemy2_display <=1;
                e2_bullet_x <= 3*DISPLAY_WIDTH/4 - 1;
                e2_bullet_y <= DISPLAY_HEIGHT+2;
                e2_bullet_display <=0;
                enemy3_x <= DISPLAY_WIDTH/2 - ENEMY3_WIDTH/2 - 1; //1024/2
-8 - 1;
                enemy3_y <= 0; //768/2 -32 -1;
                enemy3_display <=0;
                e3_bullet_x <= DISPLAY_WIDTH/2 - 1;
                e3_bullet_y <= DISPLAY_HEIGHT+2;
                e3_bullet_display <=0;
                game_over <= 0;
                p1_score <= 0;
                p2_score <= 0;
                win <= 0;
          end

          //1 PLayer Game
          if (state_packet_in == ONE_PLAYER_GAME) begin
                //Game logic changes every 60Hz
                if( (vsync!=t_vsync) && (vsync==0) )
                begin

               //ships packet
                  game_packet1_out[10:0]  <= player1_x;
               game_packet1_out[20:11] <= player1_y;
               game_packet1_out[21]    <= player1_display;
               game_packet1_out[32:22]  <= enemy1_x;
               game_packet1_out[42:33] <= enemy1_y;
               game_packet1_out[43]    <= enemy1_display;
               game_packet1_out[54:44]  <= enemy2_x;
               game_packet1_out[64:55] <= enemy2_y;
               game_packet1_out[65]    <= enemy2_display;
                  //bullets packet
                  game_packet2_out[10:0]  <= p1_bullet_x;
               game_packet2_out[20:11] <= p1_bullet_y;
               game_packet2_out[21]    <= p1_bullet_display;
               game_packet2_out[32:22]  <= e1_bullet_x;
               game_packet2_out[42:33] <= e1_bullet_y;
               game_packet2_out[43]    <= e1_bullet_display;
               game_packet2_out[54:44]  <= e2_bullet_x;
               game_packet2_out[64:55] <= e2_bullet_y;
               game_packet2_out[65]    <= e2_bullet_display;
                  //inputs
                  inputs_packet_out <= 0;

       case ({player1_display, (p1_score == 10)})

                2'b00: game_over <= 1;
```

```
                    2'b01: game_over <= 1;

                    2'b11: win <=1;

                    2'b10:      begin


                          e12_x_movement <= (e12_x_movement_counter ==179) ?
~e12_x_movement: e12_x_movement ;
                          e12_x_movement_counter <= (e12_x_movement_counter <
180) ? e12_x_movement_counter + 1 : 0;



                          //Collision detection

                          //Player1 Collisions

                            //Ships
                            if  ( enemy1_display
                                      && ( ( (enemy1_y >= player1_y) &&
(enemy1_y<= player1_y_limit) )
                                      || ( (enemy1_y_limit <= player1_y_limit)
&& (enemy1_y_limit>= player1_y) ) )
                                      && ( (enemy1_x >= player1_x) &&
(enemy1_x<= player1_x_limit)
                                      || ( (enemy1_x_limit >= player1_x) &&
(enemy1_x_limit<= player1_x_limit) ) ) ) player1_display <=0;
                            else if ( enemy2_display
                                       && ( ( (enemy2_y >= player1_y) &&
(enemy2_y<= player1_y_limit) )
                                       || ( (enemy2_y_limit <= player1_y_limit)
&& (enemy2_y_limit>= player1_y) ) )
                                       && ( ( (enemy2_x >= player1_x) &&
(enemy2_x<= player1_x_limit) )
                                       || ( (enemy2_x_limit >= player1_x) &&
(enemy2_x_limit<= player1_x_limit) ) ) ) player1_display <=0;
                                //Bullets
                            else if ( (e1_bullet_y >= player1_y) &&
(e1_bullet_y<= player1_y_limit)
                                       && (e1_bullet_x >= player1_x) &&
(e1_bullet_x<= player1_x_limit) ) begin
                                      player1_display <=0;
                                      e1_bullet_display <=0;
                            end
                            else if ( (e2_bullet_y >= player1_y) &&
(e2_bullet_y<= player1_y_limit) &&
                                                  (e2_bullet_x >= player1_x) &&
(e2_bullet_x<= player1_x_limit) ) begin
                                      player1_display <=0;
                                      e2_bullet_display <=0;
                            end


                            else begin
```

```
                              player1_display <= 1;
                              //Player1 movement code//
                                   //player1: y-direction
                                   if (up) player1_y <= ( ((player1_y -
PLAYER1_SPEED)>= (DISPLAY_HEIGHT - PLAYER1_HEIGHT -1)) || ((player1_y -
PLAYER1_SPEED)< 2 ) ) ? 0 : player1_y - PLAYER1_SPEED;
                                   else if (down) player1_y <= ((player1_y +
PLAYER1_SPEED)>=(DISPLAY_HEIGHT - PLAYER1_HEIGHT -1))

            ? (DISPLAY_HEIGHT-PLAYER1_HEIGHT-1) : player1_y + PLAYER1_SPEED;
                                   else player1_y <= player1_y;

                                   //player1: x-direction
                                   if (left) player1_x <= ( (player1_x -
PLAYER1_SPEED)>= (DISPLAY_WIDTH - PLAYER1_WIDTH -1) ) || ( (player1_x -
PLAYER1_SPEED)< 2)  ? 0 : player1_x - PLAYER1_SPEED;
                                   else if (right) player1_x <= ((player1_x
+ PLAYER1_SPEED)>=(DISPLAY_WIDTH - PLAYER1_WIDTH -1))

            ? (DISPLAY_WIDTH-PLAYER1_WIDTH-1) : player1_x + PLAYER1_SPEED;
                                   else player1_x <= player1_x;



                              //Enemy1 bullet
                              if ( enemy1_display &&
(e1_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                         e1_bullet_x <= enemy1_x +
ENEMY_WIDTH/2;
                                         e1_bullet_y <= enemy1_y +
ENEMY_HEIGHT +3;
                                         e1_bullet_display <= 1;
                                   end
                                   else begin
                                         e1_bullet_y <= (
(e1_bullet_y==DISPLAY_HEIGHT+2) || ((e1_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                         : e1_bullet_y + BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                         e1_bullet_display <= (
(e1_bullet_y==DISPLAY_HEIGHT+2) || ((e1_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                   end



                              //Enemy2 bullet

                              if ( enemy2_display &&
(e2_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                         e2_bullet_x <= enemy2_x +
ENEMY_WIDTH/2;
                                         e2_bullet_y <= enemy2_y +
ENEMY_HEIGHT +3;
                                         e2_bullet_display <= 1;
                                   end
                                   else begin
```

```
                                        e2_bullet_y <= (
(e2_bullet_y==DISPLAY_HEIGHT+2) || ((e2_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                        : e2_bullet_y + BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                        e2_bullet_display <= (
(e2_bullet_y==DISPLAY_HEIGHT+2) || ((e2_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                  end


                            end


                            //Enemy1 Collisions with p1 bullet
                        if ( enemy1_display && (p1_bullet_y >= enemy1_y)
                        && (p1_bullet_y<= enemy1_y_limit)
                        && (p1_bullet_x >= enemy1_x)
                        && (p1_bullet_x<= enemy1_x_limit) ) begin
                                  enemy1_display <=0;
                                  p1_score <= p1_score + 1;
                                  p1_bullet_display <=0;
                                  p1_bullet_y<=DISPLAY_HEIGHT+2;
                        end
                        //Enemy2 Collisions with p1 bullet
                        else if ( enemy2_display && (p1_bullet_y >=
enemy2_y)
                                        && (p1_bullet_y<= enemy2_y_limit)
                                        && (p1_bullet_x >= enemy2_x)
                                        && (p1_bullet_x<= enemy2_x_limit) )
begin
                                  enemy2_display <=0;
                                  p1_score <= p1_score + 1;
                                  p1_bullet_display <=0;
                                  p1_bullet_y<=DISPLAY_HEIGHT+2;
                            end
                        else begin
                                  p1_score <= p1_score;
            //player1 bullet:
                            //Shoot
                                  if (button_0 &&
(p1_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                        p1_bullet_x <= player1_x +
PLAYER1_WIDTH/2;
                                        p1_bullet_y <= player1_y -3;
                                        p1_bullet_display <= 1;
                                  end
                                  else begin
                                        p1_bullet_y <= (
(p1_bullet_y==DISPLAY_HEIGHT+2) || ((p1_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                        : p1_bullet_y - BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                        p1_bullet_display <= (
(p1_bullet_y==DISPLAY_HEIGHT+2) || ((p1_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                  end
```

```
                              end
                              //Enemy1 Collisions with p2 bullet
                              //if
                              //Enemy2 Collisions with p2 bullet
                              //else if
                              //else
                              //Enemy1 movement code
                              //Enemy2 movement code
                              //Player2 bullet
                              //end

                                    //State of Enemies 1 and 2

                        if( ~enemy1_display && ~enemy2_display ) begin

                              if (e12_respawn_counter == 179) begin
                                    enemy1_display <=1;
                                    enemy1_x <= DISPLAY_WIDTH/4 -
ENEMY_WIDTH/2 - 1; //1024/2 -8 - 1;
                                    enemy1_y <= 0; //768/2 -32 -1;
                                    enemy2_display <=1;
                                    enemy2_x <= 3*DISPLAY_WIDTH/4 -
ENEMY_WIDTH/2 - 1; //1024/2 -8 - 1;
                                    enemy2_y <= 0; //768/2 -32 -1;
                                    e12_respawn_counter <=0;
                              end
                              else
                                    e12_respawn_counter <=
e12_respawn_counter +1;
                        end
                        else begin
                                //Enemy1 movement code
                                if (enemy1_display) begin
                                    if (enemy1_y >= DISPLAY_HEIGHT)
enemy1_display <= 0;
                                    else begin
                                    enemy1_y <= enemy1_y + ENEMY_SPEED;
                                    enemy1_x <= e12_x_movement ? enemy1_x +
ENEMY_SPEED: enemy1_x - ENEMY_SPEED; //every 3 seconds ship changes x
orientation
                                    end
                                end
                                else begin
                                    enemy1_y <= enemy1_y;
                                    enemy1_x <= enemy1_x;
                                end

                                    //Enemy2 movement code
                                if (enemy2_display) begin
                                    if (enemy2_y >= DISPLAY_HEIGHT)
enemy2_display <= 0;
                                    else begin
                                          enemy2_y <= enemy2_y + ENEMY_SPEED;
                                          enemy2_x <= e12_x_movement ?
enemy2_x + ENEMY_SPEED: enemy2_x - ENEMY_SPEED; //every 3 seconds ship
changes x orientation
                                    end
```

```
                                end
                                else begin
                                    enemy2_y <= enemy2_y;
                                    enemy2_x <= enemy2_x;
                                end
                        end




                        end

            endcase
      end
      else
      begin
                  player1_x <= player1_x;
                  player1_y <= player1_y;
                  p1_bullet_x <= p1_bullet_x;
                  p1_bullet_y <= p1_bullet_y;
                  enemy1_x <= enemy1_x;
                  enemy1_y <= enemy1_y;
                  e1_bullet_x <= e1_bullet_x;
                  e1_bullet_y <= e1_bullet_y;
                  enemy2_x <= enemy2_x;
                  enemy2_y <= enemy2_y;
                  e2_bullet_x <= e2_bullet_x;
                  e2_bullet_y <= e2_bullet_y;
                  enemy3_x <= enemy2_x;
                  enemy3_y <= enemy2_y;
                  e3_bullet_x <= e3_bullet_x;
                  e3_bullet_y <= e3_bullet_y;
                  player1_display <= player1_display;
                  p1_bullet_display <= p1_bullet_display;
                  enemy1_display <= enemy1_display;
                  e1_bullet_display <= e1_bullet_display;
                  enemy2_display <= enemy2_display;
                  e2_bullet_display <= e2_bullet_display;
                  enemy3_display <= enemy3_display;
                  e3_bullet_display <= e3_bullet_display;
                  p1_score <= p1_score;
                  win <= win;
      end
      end

      else if (state_packet_in == TWO_PLAYER_GAME_HOST) begin
                  //2 PLayer Game
                  if( (vsync!=t_vsync) && (vsync==0) )
                  begin

                 //ships packet
                    game_packet1_out[10:0]  <= player1_x;
                 game_packet1_out[20:11] <= player1_y;
                 game_packet1_out[21]    <= player1_display;
                 game_packet1_out[32:22]  <= enemy1_x;
                 game_packet1_out[42:33] <= enemy1_y;
```

```
                    game_packet1_out[43]    <= enemy1_display;
                    game_packet1_out[54:44]  <= enemy2_x;
                    game_packet1_out[64:55] <= enemy2_y;
                    game_packet1_out[65]    <= enemy2_display;
                       game_packet1_out[76:66]  <= player2_x;
                    game_packet1_out[86:77] <= player2_y;
                    game_packet1_out[87]    <= player2_display;
                       game_packet1_out[98:88]  <= enemy3_x;
                    game_packet1_out[108:99] <= enemy3_y;
                    game_packet1_out[109]    <= enemy3_display;
                       //scores
                       game_packet1_out[114:110]    <= p1_score;
                       game_packet1_out[119:115]    <= p2_score;
                       //bullets packet
                       game_packet2_out[10:0]  <= p1_bullet_x;
                    game_packet2_out[20:11] <= p1_bullet_y;
                    game_packet2_out[21]    <= p1_bullet_display;
                    game_packet2_out[32:22]  <= e1_bullet_x;
                    game_packet2_out[42:33] <= e1_bullet_y;
                    game_packet2_out[43]    <= e1_bullet_display;
                    game_packet2_out[54:44]  <= e2_bullet_x;
                    game_packet2_out[64:55] <= e2_bullet_y;
                    game_packet2_out[65]    <= e2_bullet_display;
                       game_packet2_out[76:66]  <= p2_bullet_x;
                    game_packet2_out[86:77] <= p2_bullet_y;
                    game_packet2_out[87]    <= p2_bullet_display;
                       game_packet2_out[98:88]  <= e3_bullet_x;
                    game_packet2_out[108:99] <= e3_bullet_y;
                    game_packet2_out[109]    <= e3_bullet_display;
                    //ready signal
                       inputs_packet_out <= 0;

          case ( {(player1_display || player2_display) , (p1_score+p2_score ==
20)} )

                     2'b00: game_over <= 1;

                     2'b01: game_over <= 1;

                     2'b11: win <=1;

                     2'b10:      begin


                           e12_x_movement <= (e12_x_movement_counter ==179) ?
~e12_x_movement: e12_x_movement ;
                           e12_x_movement_counter <= (e12_x_movement_counter <
180) ? e12_x_movement_counter + 1 : 0;

                           e3_x_movement <= (e3_x_movement_counter ==89) ?
~e3_x_movement: e3_x_movement ;
                           e3_x_movement_counter <= (e3_x_movement_counter < 90)
? e3_x_movement_counter + 1 : 0;
```

```
                                //Collision detection

                        //Player1 with Ships
                          if  ( enemy1_display
                                        && ( ( (enemy1_y >= player1_y) &&
(enemy1_y<= player1_y_limit) )
                                        || ( (enemy1_y_limit <= player1_y_limit)
&& (enemy1_y_limit>= player1_y) ) )
                                        && ( (enemy1_x >= player1_x) &&
(enemy1_x<= player1_x_limit)
                                        || ( (enemy1_x_limit >= player1_x) &&
(enemy1_x_limit<= player1_x_limit) ) ) ) player1_display <=0;
                          else if  ( enemy2_display
                                        && ( ( (enemy2_y >= player1_y) &&
(enemy2_y<= player1_y_limit) )
                                        || ( (enemy2_y_limit <= player1_y_limit)
&& (enemy2_y_limit>= player1_y) ) )
                                        && ( (enemy2_x >= player1_x) &&
(enemy2_x<= player1_x_limit)
                                        || ( (enemy2_x_limit >= player1_x) &&
(enemy2_x_limit<= player1_x_limit) ) ) ) player1_display <=0;
                          else if  ( enemy3_display
                                        && ( ( (enemy3_y >= player1_y) &&
(enemy3_y<= player1_y_limit) )
                                        || ( (enemy3_y_limit <= player1_y_limit)
&& (enemy3_y_limit>= player1_y) ) )
                                        && ( (enemy3_x >= player1_x) &&
(enemy3_x<= player1_x_limit)
                                        || ( (enemy3_x_limit >= player1_x) &&
(enemy3_x_limit<= player1_x_limit) ) ) ) player1_display <=0;
                          else begin

                                 //Player1 movement code//
                                        //player1: y-direction
                                        if (up) player1_y <= ( ((player1_y -
PLAYER1_SPEED)>= (DISPLAY_HEIGHT - PLAYER1_HEIGHT -1)) || ((player1_y -
PLAYER1_SPEED)< 2 ) ) ? 0 : player1_y - PLAYER1_SPEED;
                                        else if (down) player1_y <= ((player1_y +
PLAYER1_SPEED)>=(DISPLAY_HEIGHT - PLAYER1_HEIGHT -1))

            ? (DISPLAY_HEIGHT-PLAYER1_HEIGHT-1) : player1_y + PLAYER1_SPEED;
                                        else player1_y <= player1_y;

                                        //player1: x-direction
                                        if (left) player1_x <= ( (player1_x -
PLAYER1_SPEED)>= (DISPLAY_WIDTH - PLAYER1_WIDTH -1) ) || ( (player1_x -
PLAYER1_SPEED)< 2)  ? 0 : player1_x - PLAYER1_SPEED;
                                        else if (right) player1_x <= ((player1_x
+ PLAYER1_SPEED)>=(DISPLAY_WIDTH - PLAYER1_WIDTH -1))

            ? (DISPLAY_WIDTH-PLAYER1_WIDTH-1) : player1_x + PLAYER1_SPEED;
                                        else player1_x <= player1_x;
                                end


                        //Player2 Collisions    with Ships
```

```
                          if  ( enemy1_display
                                    && ( ( (enemy1_y >= player2_y) &&
(enemy1_y<= player2_y_limit) )
                                    || ( (enemy1_y_limit <= player2_y_limit)
&& (enemy1_y_limit>= player2_y) ) )
                                    && ( (enemy1_x >= player2_x) &&
(enemy1_x<= player2_x_limit)
                                    || ( (enemy1_x_limit >= player2_x) &&
(enemy1_x_limit<= player2_x_limit) ) ) ) player2_display <=0;

                          else if( enemy2_display
                                    && ( ( (enemy2_y >= player2_y) &&
(enemy2_y<= player2_y_limit) )
                                    || ( (enemy2_y_limit <= player2_y_limit)
&& (enemy2_y_limit>= player2_y) ) )
                                    && ( (enemy2_x >= player2_x) &&
(enemy2_x<= player2_x_limit)
                                    || ( (enemy2_x_limit >= player2_x) &&
(enemy2_x_limit<= player2_x_limit) ) ) ) player2_display <=0;
                          else if  ( enemy3_display
                                    && ( ( (enemy3_y >= player2_y) &&
(enemy3_y<= player2_y_limit) )
                                    || ( (enemy3_y_limit <= player2_y_limit)
&& (enemy3_y_limit>= player2_y) ) )
                                    && ( (enemy3_x >= player2_x) &&
(enemy3_x<= player2_x_limit)
                                    || ( (enemy3_x_limit >= player2_x) &&
(enemy3_x_limit<= player2_x_limit) ) ) ) player2_display <=0;
                          else begin
                               //player2 movement code//
                                    //player2: y-direction
                                    if (ext_up) player2_y <= ( ((player2_y -
PLAYER2_SPEED)>= (DISPLAY_HEIGHT - PLAYER2_HEIGHT -1)) || ((player2_y -
PLAYER2_SPEED)< 2 ) ) ? 0 : player2_y - PLAYER2_SPEED;
                                    else if (ext_down) player2_y <=
((player2_y + PLAYER2_SPEED)>=(DISPLAY_HEIGHT - PLAYER2_HEIGHT -1))

            ? (DISPLAY_HEIGHT-PLAYER2_HEIGHT-1) : player2_y + PLAYER2_SPEED;
                                    else player2_y <= player2_y;

                                    //player2: x-direction
                                    if (ext_left) player2_x <= ( (player2_x -
PLAYER2_SPEED)>= (DISPLAY_WIDTH - PLAYER2_WIDTH -1) ) || ( (player2_x -
PLAYER2_SPEED)< 2)  ? 0 : player2_x - PLAYER2_SPEED;
                                    else if (ext_right) player2_x <=
((player2_x + PLAYER2_SPEED)>=(DISPLAY_WIDTH - PLAYER2_WIDTH -1))

            ? (DISPLAY_WIDTH-PLAYER2_WIDTH-1) : player2_x + PLAYER2_SPEED;
                                    else player2_x <= player2_x;
                              end



                              //Player1 Collisions with bullets
                          if ( (e1_bullet_y >= player1_y) && (e1_bullet_y<=
player1_y_limit)
```

```
                                    && (e1_bullet_x >= player1_x) &&
(e1_bullet_x<= player1_x_limit) ) begin
                                    player1_display <=0;
                                    e1_bullet_display <=0;
                          end
                              //Player2 Collisions with e1_bullet
                          else if( (e1_bullet_y >= player2_y) &&
(e1_bullet_y<= player2_y_limit)
                                    && (e1_bullet_x >= player2_x) &&
(e1_bullet_x<= player2_x_limit) )   begin
                                    player2_display <=0;
                                    e1_bullet_display <=0;
                          end
                          else begin
                                    //Enemy1 bullet
                                    if ( enemy1_display &&
(e1_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                          e1_bullet_x <= enemy1_x +
ENEMY_WIDTH/2;
                                          e1_bullet_y <= enemy1_y +
ENEMY_HEIGHT +3;
                                          e1_bullet_display <= 1;
                                    end
                                    else begin
                                          e1_bullet_y <= (
(e1_bullet_y==DISPLAY_HEIGHT+2) || ((e1_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                          : e1_bullet_y + BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                          e1_bullet_display <= (
(e1_bullet_y==DISPLAY_HEIGHT+2) || ((e1_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                    end
                          end

                              //Player1 collisions with e2 bullet
                          if ( (e2_bullet_y >= player1_y) && (e2_bullet_y<=
player1_y_limit) &&
                                          (e2_bullet_x >= player1_x) &&
(e2_bullet_x<= player1_x_limit) ) begin
                                    player1_display <=0;
                                    e2_bullet_display <=0;
                          end
                          //Player2 Collisions with e2_bullet
                          else if ( (e2_bullet_y >= player2_y) &&
(e2_bullet_y<= player2_y_limit) &&
                                          (e2_bullet_x >= player2_x) &&
(e2_bullet_x<= player2_x_limit) ) begin
                                    player2_display <=0;
                                    e2_bullet_display <=0;
                          end
                          else begin
                              //Enemy2 bullet

                                    if ( enemy2_display &&
(e2_bullet_y==DISPLAY_HEIGHT+2) ) begin
```

```
                                                  e2_bullet_x <= enemy2_x +
ENEMY_WIDTH/2;
                                                  e2_bullet_y <= enemy2_y +
ENEMY_HEIGHT +3;
                                                  e2_bullet_display <= 1;
                                            end
                                            else begin
                                                  e2_bullet_y <= (
(e2_bullet_y==DISPLAY_HEIGHT+2) || ((e2_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                                  : e2_bullet_y + BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                                  e2_bullet_display <= (
(e2_bullet_y==DISPLAY_HEIGHT+2) || ((e2_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                            end
                                      end

                                      //Player1 collisions with e3 bullet
                                  if ( (e3_bullet_y >= player1_y) && (e3_bullet_y<=
player1_y_limit) &&
                                                        (e3_bullet_x >= player1_x) &&
(e3_bullet_x<= player1_x_limit) ) begin
                                            player1_display <=0;
                                            e3_bullet_display <=0;
                                  end
                                  //Player2 Collisions with e3_bullet
                                  else if ( (e3_bullet_y >= player2_y) &&
(e3_bullet_y<= player2_y_limit) &&
                                                        (e3_bullet_x >= player2_x) &&
(e3_bullet_x<= player2_x_limit) ) begin
                                            player2_display <=0;
                                            e3_bullet_display <=0;
                                  end
                                  else begin
                                      //Enemy3 bullet

                                            if ( enemy3_display &&
(e3_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                                  e3_bullet_x <= enemy3_x +
ENEMY3_WIDTH/2;
                                                  e3_bullet_y <= enemy3_y +
ENEMY3_HEIGHT +3;
                                                  e3_bullet_display <= 1;
                                            end
                                            else begin
                                                  e3_bullet_y <= (
(e3_bullet_y==DISPLAY_HEIGHT+2) || ((e3_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                                                  : e3_bullet_y + BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                                  e3_bullet_display <= (
(e3_bullet_y==DISPLAY_HEIGHT+2) || ((e3_bullet_y +
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                            end
                                      end
```

```
            //Enemy1 Collisions with p1 bullet
        if ( enemy1_display && (p1_bullet_y >= enemy1_y)
        && (p1_bullet_y<= enemy1_y_limit)
        && (p1_bullet_x >= enemy1_x)
        && (p1_bullet_x<= enemy1_x_limit) ) begin
                enemy1_display <=0;
                p1_score <= p1_score + 1;
                p1_bullet_display <=0;
                p1_bullet_y<=DISPLAY_HEIGHT+2;
        end
        //Enemy2 Collisions with p1 bullet
        else if ( enemy2_display && (p1_bullet_y >=
enemy2_y)
                        && (p1_bullet_y<= enemy2_y_limit)
                        && (p1_bullet_x >= enemy2_x)
                        && (p1_bullet_x<= enemy2_x_limit) )
begin
                enemy2_display <=0;
                p1_score <= p1_score + 1;
                p1_bullet_display <=0;
                p1_bullet_y<=DISPLAY_HEIGHT+2;
            end
        //Enemy3 Collisions with p1 bullet
        else if (enemy3_display && (p1_bullet_y >=
enemy3_y)
                        && (p1_bullet_y<= enemy3_y_limit)
                        && (p1_bullet_x >= enemy3_x)
                        && (p1_bullet_x<= enemy3_x_limit) )
begin
                enemy3_display <=0;
                p1_score <= p1_score + 2;
                p1_bullet_display <=0;
                p1_bullet_y<=DISPLAY_HEIGHT+2;
            end
        else begin
                p1_score <= p1_score;

    //player1 bullet:
            //Shoot
                if (player1_display && button_0 &&
(p1_bullet_y==DISPLAY_HEIGHT+2) ) begin
                        p1_bullet_x <= player1_x +
PLAYER1_WIDTH/2;
                        p1_bullet_y <= player1_y -3;
                        p1_bullet_display <= 1;
                end
                else begin
                        p1_bullet_y <= (
(p1_bullet_y==DISPLAY_HEIGHT+2) || ((p1_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
                        : p1_bullet_y - BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
```

```
                                          p1_bullet_display <= (
(p1_bullet_y==DISPLAY_HEIGHT+2) || ((p1_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                  end
                          end


                          //Enemy1 Collisions with p2 bullet
                     if ( enemy1_display && (p2_bullet_y >= enemy1_y)
                     && (p2_bullet_y<= enemy1_y_limit)
                     && (p2_bullet_x >= enemy1_x)
                     && (p2_bullet_x<= enemy1_x_limit) ) begin
                                  enemy1_display <=0;
                                  p2_score <= p2_score + 1;
                                  p2_bullet_display <=0;
                                  p2_bullet_y<=DISPLAY_HEIGHT+2;
                     end
                     //Enemy2 Collisions with p2 bullet
                     else if ( enemy2_display && (p2_bullet_y >=
enemy2_y)
                                          && (p2_bullet_y<= enemy2_y_limit)
                                          && (p2_bullet_x >= enemy2_x)
                                          && (p2_bullet_x<= enemy2_x_limit) )
begin
                                  enemy2_display <=0;
                                  p2_score <= p2_score + 1;
                                  p2_bullet_display <=0;
                                  p2_bullet_y<=DISPLAY_HEIGHT+2;
                          end
                          //Enemy3 Collisions with p2 bullet
                     else if ( enemy3_display && (p2_bullet_y >=
enemy3_y)
                                          && (p2_bullet_y<= enemy3_y_limit)
                                          && (p2_bullet_x >= enemy3_x)
                                          && (p2_bullet_x<= enemy3_x_limit) )
begin
                                  enemy3_display <=0;
                                  p2_score <= p2_score + 2;
                                  p2_bullet_display <=0;
                                  p2_bullet_y<=DISPLAY_HEIGHT+2;
                          end
                     else begin
                                  p2_score <= p2_score;

            //player2 bullet:
                          //Shoot
                                  if (player2_display && ext_button_0 &&
(p2_bullet_y==DISPLAY_HEIGHT+2) ) begin
                                          p2_bullet_x <= player2_x +
PLAYER2_WIDTH/2;
                                          p2_bullet_y <= player2_y -3;
                                          p2_bullet_display <= 1;
                                  end
                                  else begin
                                          p2_bullet_y <= (
(p2_bullet_y==DISPLAY_HEIGHT+2) || ((p2_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? DISPLAY_HEIGHT+2
```

```
                                          : p2_bullet_y - BULLET_SPEED;
//DISPLAY_HEIGHT+2 TO HIDE IT
                                          p2_bullet_display <= (
(p2_bullet_y==DISPLAY_HEIGHT+2) || ((p2_bullet_y -
BULLET_SPEED)>=DISPLAY_HEIGHT) ) ? 0: 1;
                                    end
                              end


                        //State of Enemy1 and Enemy2
                        if( ~enemy1_display && ~enemy2_display ) begin

                              if (e12_respawn_counter == 179) begin
                                    enemy1_display <=1;
                                    enemy1_x <= DISPLAY_WIDTH/4 -
ENEMY_WIDTH/2 - 1; //1024/2 -8 - 1;
                                    enemy1_y <= 0; //768/2 -32 -1;
                                    enemy2_display <=1;
                                    enemy2_x <= 3*DISPLAY_WIDTH/4 -
ENEMY_WIDTH/2 - 1; //1024/2 -8 - 1;
                                    enemy2_y <= 0; //768/2 -32 -1;
                                    e12_respawn_counter <=0;
                              end
                              else
                                    e12_respawn_counter <=
e12_respawn_counter +1;
                        end
                        else begin
                                //Enemy1 movement code
                                if (enemy1_display) begin
                                    if (enemy1_y >= DISPLAY_HEIGHT)
enemy1_display <= 0;
                                    else begin
                                    enemy1_y <= enemy1_y + ENEMY_SPEED;
                                    enemy1_x <= e12_x_movement ? enemy1_x +
ENEMY_SPEED: enemy1_x - ENEMY_SPEED; //every 3 seconds ship changes x
orientation
                                    end
                                end
                                else begin
                                    enemy1_y <= enemy1_y;
                                    enemy1_x <= enemy1_x;
                                end

                                    //Enemy2 movement code
                                if (enemy2_display) begin
                                    if (enemy2_y >= DISPLAY_HEIGHT)
enemy2_display <= 0;
                                    else begin
                                          enemy2_y <= enemy2_y + ENEMY_SPEED;
                                          enemy2_x <= e12_x_movement ?
enemy2_x + ENEMY_SPEED: enemy2_x - ENEMY_SPEED; //every 3 seconds ship
changes x orientation
                                    end
                                end
                                else begin
                                    enemy2_y <= enemy2_y;
```

```
                                    enemy2_x <= enemy2_x;
                                end
                        end

                  //State of Enemy 3

                        if( ~enemy3_display) begin

                              if (e3_respawn_counter == 239) begin
                                    enemy3_display <=1;
                                    enemy3_x <= DISPLAY_WIDTH/2 -
ENEMY_WIDTH/2 - 1; //1024/2 -8 - 1;
                                    enemy3_y <= 0; //768/2 -32 -1;
                                    e3_respawn_counter <=0;
                              end
                              else
                                    e3_respawn_counter <= e3_respawn_counter
+1;
                        end
                        else begin
                                //Enemy3 movement code
                                if (enemy3_display) begin
                                    if (enemy3_y >= DISPLAY_HEIGHT)
enemy3_display <= 0;
                                    else begin
                                    enemy3_y <= enemy3_y + ENEMY3_SPEED;
                                    enemy3_x <= e3_x_movement ? enemy3_x +
ENEMY3_SPEED: enemy3_x - ENEMY3_SPEED; //every 1.5 seconds ship changes x
orientation
                                    end
                                end
                                else begin
                                    enemy3_y <= enemy3_y;
                                    enemy3_x <= enemy3_x;
                                end
                        end






                        end
                  endcase
      end
      else
      begin
                  player1_x <= player1_x;
                  player1_y <= player1_y;
                  p1_bullet_x <= p1_bullet_x;
                  p1_bullet_y <= p1_bullet_y;
                  player2_x <= player2_x;
                  player2_y <= player2_y;
                  p2_bullet_x <= p2_bullet_x;
                  p2_bullet_y <= p2_bullet_y;
                  enemy1_x <= enemy1_x;
                  enemy1_y <= enemy1_y;
```

```
                e1_bullet_x <= e1_bullet_x;
                e1_bullet_y <= e1_bullet_y;
                enemy2_x <= enemy2_x;
                enemy2_y <= enemy2_y;
                e2_bullet_x <= e2_bullet_x;
                e2_bullet_y <= e2_bullet_y;
                enemy3_x <= enemy3_x;
                enemy3_y <= enemy3_y;
                e3_bullet_x <= e3_bullet_x;
                e3_bullet_y <= e3_bullet_y;
                player1_display <= player1_display;
                p1_bullet_display <= p1_bullet_display;
                player2_display <= player2_display;
                p2_bullet_display <= p2_bullet_display;
                enemy1_display <= enemy1_display;
                e1_bullet_display <= e1_bullet_display;
                enemy2_display <= enemy2_display;
                e2_bullet_display <= e2_bullet_display;
                enemy3_display <= enemy3_display;
                e3_bullet_display <= e3_bullet_display;
                p1_score <= p1_score;
                p2_score <= p2_score;
                win <= win;
    end



    end

    else if (state_packet_in == TWO_PLAYER_GAME_CLIENT) begin

                if( (vsync!=t_vsync) && (vsync==0) )
                begin
                      game_packet1_out <= 0;
                      game_packet2_out <= 0;
                      inputs_packet_out[0] <= button_0;
                      inputs_packet_out[1] <= up;
                      inputs_packet_out[2] <= down;
                      inputs_packet_out[3] <= left;
                      inputs_packet_out[4] <= right;
                      game_over <= 0;
                      win <= 0;
                end

    end

    else begin
            game_packet1_out <= 0;
            game_packet2_out <= 0;
            inputs_packet_out <= 0;
            game_over <= 0;
            win <= 0;
    end

    end
endmodule
```

F. Video Controller

Pixel Generator

```
//////////////////////////////////////////////////////////////////////////////
///
//
// The pixel generator
//
//////////////////////////////////////////////////////////////////////////////
///

module pixel_gen #(parameter DISPLAY_WIDTH = 1024,
                                                        DISPLAY_HEIGHT = 768,
                                                        P1_BULLET_RADIUS = 2,
                                                        P1_BULLET_COLOR =
3'b111,//DEFAULT: WHITE
                                                        PLAYER1_WIDTH = 36,
// default width: 32 pixels
                                                        PLAYER1_HEIGHT = 54,
// default height: 32 pixels
                                                        PLAYER1_COLOR =
3'b110,
                                                        PLAYER1_SPEED = 4,
                                                        P2_BULLET_RADIUS = 2,
                                                        P2_BULLET_COLOR =
3'b111,//DEFAULT: WHITE
                                                        PLAYER2_WIDTH = 36,
// default width: 32 pixels
                                                        PLAYER2_HEIGHT = 54,
// default height: 32 pixels
                                                        PLAYER2_COLOR =
3'b011,
                                                        PLAYER2_SPEED = 4,
                                                        E_BULLET_RADIUS = 2,
                                                        E_BULLET_COLOR =
3'b101,//DEFAULT: PINK
                                                        ENEMY_COLOR = 3'b100,
                                                        ENEMY_WIDTH = 64,
// default width: 32 pixels
                                                        ENEMY_HEIGHT = 64,
                                                        ENEMY_SPEED = 1,
                                    // default height: 32 pixels
                                                        BULLET_SPEED = 12,
                                                        ENEMY3_COLOR =
3'b101,
                                                        ENEMY3_WIDTH = 128,
// default width: 32 pixels
                                                        ENEMY3_HEIGHT = 64,
                                                        ENEMY3_SPEED = 2,
                                    // default height: 32 pixels
                                                        E3_BULLET_RADIUS = 4,
                                                        E3_BULLET_COLOR =
3'b100)
```

```
(input vclock,  // 65MHz clock
 input reset,         // 1 to initialize module
 input [10:0] hcount, // horizontal index of current pixel (0..1023)
 input [9:0]    vcount, // vertical index of current pixel (0..767)
    input hsync,            // XVGA horizontal sync signal (active low)
 input vsync,         // XVGA vertical sync signal (active low)
 input blank,         // XVGA blanking (1 means output black pixel)
    input [127:0] game_packet1_in,
 input [127:0] game_packet2_in,
    input [79:0] id_packet_in,
    input [79:0] id,
 input [3:0] state,
    input [4:0] p1_score,
    input [4:0] p2_score,
 output reg phsync,    // pong game's horizontal sync
 output reg pvsync,    // pong game's vertical sync
 output reg pblank,    // pong game's blanking

    //
    //output reg [23:0] score1_cstring,
 output reg [2:0] pixel     // pong game's pixel
 );

localparam START                   = 4'd0;
localparam LOGIN                   = 4'd1;
localparam ONE_PLAYER              = 4'd2;
localparam ONE_PLAYER_GAME         = 4'd3;
localparam TWO_PLAYERS             = 4'd4;
localparam HOST_READY_WAITING      = 4'd5;
localparam HOST_READY              = 4'd6;
localparam TWO_PLAYER_GAME_HOST    = 4'd7;
localparam TWO_PLAYER_GAME_CLIENT  = 4'd8;
localparam GAME_OVER               = 4'd9;
localparam YOU_WIN                 = 4'd10;



    reg t_vsync;
    reg t_hsync;
    reg t_blank;

    reg t2_vsync;
    reg t2_hsync;
    reg t2_blank;

    reg t3_vsync;
    reg t3_hsync;
    reg t3_blank;

    always @ (posedge vclock) begin
    t_vsync <= t3_vsync;
    t_hsync <= t3_hsync;
    t_blank <= t3_blank;
    end
    always @ (posedge vclock) begin
          t3_vsync <= t2_vsync;
          t3_hsync <= t2_hsync;
```

```
            t3_blank <= t2_blank;
      end
      always @ (posedge vclock) begin
            t2_vsync <= vsync;
            t2_hsync <=hsync;
            t2_blank <= blank;
      end




//Player 1 coordinates and pixel info
      reg [10:0] player1_x;
      reg [9:0]  player1_y;
      wire [2:0] player1_pixel;
      reg player1_display;
//P1 bullet coordinates and pixel info
      reg [10:0] p1_bullet_x;
      reg [9:0]  p1_bullet_y;
      wire [2:0] p1_bullet_pixel;
      reg p1_bullet_display;

//Player 2 coordinates and pixel info
      reg [9:0]  player2_y;
      reg [10:0] player2_x;
      wire [2:0] player2_pixel;
      reg player2_display;
//P2 bullet coordinates and pixel info
      reg [9:0]  p2_bullet_y;
      reg [10:0] p2_bullet_x;
      wire [2:0] p2_bullet_pixel;
      reg p2_bullet_display;

//Enemy 1 coordinates and pixel info
      reg [10:0] enemy1_x;
      reg [9:0]  enemy1_y;
      wire [2:0] enemy1_pixel;
      reg enemy1_display;
//E1 bullet coordinates and pixel info
      reg [10:0] e1_bullet_x;
      reg [9:0]  e1_bullet_y;
      wire [2:0] e1_bullet_pixel;
      reg e1_bullet_display;

//Enemy 2 coordinates and pixel info
      reg [10:0] enemy2_x;
      reg [9:0]  enemy2_y;
      wire [2:0] enemy2_pixel;
      reg enemy2_display;
//E2 bullet coordinates and pixel info
      reg [10:0] e2_bullet_x;
      reg [9:0]  e2_bullet_y;
      wire [2:0] e2_bullet_pixel;
      reg e2_bullet_display;

      //Enemy 3 coordinates and pixel info
```

```
      reg [10:0] enemy3_x;
      reg [9:0]  enemy3_y;
      wire [2:0] enemy3_pixel;
      reg enemy3_display;
//E3 bullet coordinates and pixel info
      reg [10:0] e3_bullet_x;
      reg [9:0]  e3_bullet_y;
      wire [2:0] e3_bullet_pixel;
      reg e3_bullet_display;




//scores

reg p1_score_in;
reg p2_score_in;


//Assign x,y coordinates and display bits from the game packets inputs
      always @ ( posedge vclock ) begin
      //ships
            if( (vsync!=t_vsync) && (vsync==0) ) begin
              player1_x = game_packet1_in[10:0];
              player1_y = game_packet1_in[20:11];
              player1_display = game_packet1_in[21];
              enemy1_x = game_packet1_in[32:22];
              enemy1_y = game_packet1_in[42:33];
              enemy1_display = game_packet1_in[43];
              enemy2_x = game_packet1_in[54:44];
              enemy2_y = game_packet1_in[64:55];
              enemy2_display = game_packet1_in[65];
              player2_x = game_packet1_in[76:66];
              player2_y = game_packet1_in[86:77];
              player2_display = game_packet1_in[87];
              enemy3_x = game_packet1_in[98:88];
              enemy3_y = game_packet1_in[108:99];
              enemy3_display = game_packet1_in[109];
              //scores
              p1_score_in = game_packet1_in[114:110];
              p2_score_in = game_packet1_in[119:115];
            //bullets
              p1_bullet_x = game_packet2_in[10:0];
              p1_bullet_y = game_packet2_in[20:11];
              p1_bullet_display = game_packet2_in[21];
              e1_bullet_x = game_packet2_in[32:22];
              e1_bullet_y = game_packet2_in[42:33];
              e1_bullet_display = game_packet2_in[43];
              e2_bullet_x = game_packet2_in[54:44];
              e2_bullet_y = game_packet2_in[64:55];
              e2_bullet_display = game_packet2_in[65];
              p2_bullet_x = game_packet2_in[76:66];
              p2_bullet_y = game_packet2_in[86:77];
              p2_bullet_display = game_packet2_in[87];
              e3_bullet_x = game_packet2_in[98:88];
              e3_bullet_y = game_packet2_in[108:99];
              e3_bullet_display = game_packet2_in[109];
```

```
            //state
             // state = state_packet_in[3:0];
            //  id = id_packet_in[1:0];

            end
      end

//Player 1 blob
ship_sprite
player1(.vclock(vclock),.display(player1_display),.x(player1_x),.y(player1_y)
,
.hcount(hcount),.vcount(vcount),
      .pixel(player1_pixel));

//    blob
#(.WIDTH(PLAYER1_WIDTH),.HEIGHT(PLAYER1_HEIGHT),.COLOR(PLAYER1_COLOR))   //
      //
      player1(.x(player1_x),.y(player1_y),.hcount(hcount),.vcount(vcount),
.display(player1_display),
      //            .pixel(player1_pixel));
//P1 Bullet
      circle #(.RADIUS(P1_BULLET_RADIUS),.COLOR(P1_BULLET_COLOR))

      p1_bullet(.vclock(vclock),.x(p1_bullet_x),.y(p1_bullet_y),.hcount(hcoun
t),.vcount(vcount),
            .display(p1_bullet_display), .pixel(p1_bullet_pixel));

//Player 2 blob
ship_sprite2
player2(.vclock(vclock),.display(player2_display),.x(player2_x),.y(player2_y)
,.hcount(hcount),.vcount(vcount),
             .pixel(player2_pixel));



//    blob
#(.WIDTH(PLAYER2_WIDTH),.HEIGHT(PLAYER2_HEIGHT),.COLOR(PLAYER2_COLOR))   //
//
      player2(.x(player2_x),.y(player2_y),.hcount(hcount),.vcount(vcount),
.display(player2_display),
//                  .pixel(player2_pixel));
//P2 Bullet
      circle #(.RADIUS(P2_BULLET_RADIUS),.COLOR(P2_BULLET_COLOR))

      p2_bullet(.vclock(vclock),.x(p2_bullet_x),.y(p2_bullet_y),.hcount(hcoun
t),.vcount(vcount),
            .display(p2_bullet_display), .pixel(p2_bullet_pixel));

//Enemy 1 blob
enemy1_sprite
enemy1(.vclock(vclock),.display(enemy1_display),.x(enemy1_x),.y(enemy1_y),.hc
ount(hcount),.vcount(vcount),
             .pixel(enemy1_pixel));
```

```
//    blob
#(.WIDTH(PLAYER1_WIDTH),.HEIGHT(PLAYER1_HEIGHT),.COLOR(ENEMY_COLOR))    //
//     enemy1(.x(enemy1_x),.y(enemy1_y),.hcount(hcount),.vcount(vcount),
.display(enemy1_display),
//             .pixel(enemy1_pixel));

//E1 Bullet
      circle #(.RADIUS(E_BULLET_RADIUS),.COLOR(E_BULLET_COLOR))

      e1_bullet(.vclock(vclock),.x(e1_bullet_x),.y(e1_bullet_y),.hcount(hcoun
t),.vcount(vcount),
                  .display(e1_bullet_display), .pixel(e1_bullet_pixel));

//Enemy 2 blob
enemy2_sprite
enemy2(.vclock(vclock),.display(enemy2_display),.x(enemy2_x),.y(enemy2_y),.hc
ount(hcount),.vcount(vcount),
             .pixel(enemy2_pixel));

//    blob
#(.WIDTH(PLAYER1_WIDTH),.HEIGHT(PLAYER1_HEIGHT),.COLOR(ENEMY_COLOR))    //
//     enemy2(.x(enemy2_x),.y(enemy2_y),.hcount(hcount),.vcount(vcount),
.display(enemy2_display),
//             .pixel(enemy2_pixel));
//E2 Bullet
      circle #(.RADIUS(E_BULLET_RADIUS),.COLOR(E_BULLET_COLOR))

      e2_bullet(.vclock(vclock),.x(e2_bullet_x),.y(e2_bullet_y),.hcount(hcoun
t),.vcount(vcount),
                  .display(e2_bullet_display), .pixel(e2_bullet_pixel));
//Enemy 3 blob
enemy3_sprite
enemy3(.vclock(vclock),.display(enemy3_display),.x(enemy3_x),.y(enemy3_y),.hc
ount(hcount),.vcount(vcount),
             .pixel(enemy3_pixel));



//    blob
#(.WIDTH(ENEMY3_WIDTH),.HEIGHT(ENEMY3_HEIGHT),.COLOR(ENEMY3_COLOR))    //
//     enemy3(.x(enemy3_x),.y(enemy3_y),.hcount(hcount),.vcount(vcount),
.display(enemy3_display),
//             .pixel(enemy3_pixel));
//E3 Bullet
      circle #(.RADIUS(E3_BULLET_RADIUS),.COLOR(E3_BULLET_COLOR))

      e3_bullet(.vclock(vclock),.x(e3_bullet_x),.y(e3_bullet_y),.hcount(hcoun
t),.vcount(vcount),
                  .display(e3_bullet_display), .pixel(e3_bullet_pixel));


//Character String
wire [2:0] start_pixel;
reg [135:0] start_cstring;

char_string_display #(.NCHAR(17), .NCHAR_BITS(5))
```

```
      start(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(start_cstring),.cx(DISPLAY_WIDTH/2-136),.cy(DISPLAY_HEIGHT/2-120),
                              .pixel(start_pixel) );

wire [2:0] start2_pixel;
reg [79:0] start2_cstring;


char_string_display #(.NCHAR(10), .NCHAR_BITS(4))

      start2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(start2_cstring),.cx(DISPLAY_WIDTH/2-136),.cy(DISPLAY_HEIGHT/2-120),
                              .pixel(start2_pixel) );


wire [2:0] input_pixel;
reg [175:0] input_cstring;


char_string_display #(.NCHAR(22), .NCHAR_BITS(5))

      user_input1(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(input_cstring),.cx(DISPLAY_WIDTH/2-176),.cy(DISPLAY_HEIGHT/2-24-6),
                              .pixel(input_pixel) );

wire [2:0] input2_pixel;
reg [183:0] input2_cstring;

char_string_display #(.NCHAR(23), .NCHAR_BITS(5))

      user_input2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(input2_cstring),.cx(DISPLAY_WIDTH/2-184),.cy(DISPLAY_HEIGHT/2+6),
                              .pixel(input2_pixel) );

wire [2:0] message_pixel;
reg [239:0] message_cstring;

char_string_display #(.NCHAR(30), .NCHAR_BITS(5))

      message(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(message_cstring),.cx(DISPLAY_WIDTH/2-240),.cy(DISPLAY_HEIGHT/2-60),
                              .pixel(message_pixel) );


wire [2:0] name1_pixel;
reg [79:0] name1_cstring;


char_string_display #(.NCHAR(10), .NCHAR_BITS(4))

      name1(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(name1_cstring),.cx(0),.cy(0),
                              .pixel(name1_pixel) );

wire [2:0] name2_pixel;
reg [79:0] name2_cstring;
```

```
char_string_display #(.NCHAR(10), .NCHAR_BITS(4))

      name2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(name2_cstring),.cx(DISPLAY_WIDTH-160),.cy(0),
                              .pixel(name2_pixel) );

wire [2:0] score1_pixel;
reg [47:0] score1_cstring;


char_string_display #(.NCHAR(6), .NCHAR_BITS(3))

      score1(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(score1_cstring),.cx(0),.cy(30),
                              .pixel(score1_pixel) );

wire [2:0] score2_pixel;
reg [47:0] score2_cstring;


char_string_display #(.NCHAR(6), .NCHAR_BITS(3))

      score2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
.cstring(score2_cstring),.cx(DISPLAY_WIDTH-96),.cy(30),
                              .pixel(score2_pixel) );


reg [26:0] counter;
reg [1:0] num_of_dots;

      //keep track of previous state of vsync
//one-clock-cycle-behind VGA signals

      //assign the pixel, blanking and syncs at the positive edge of the
clock
      always @ (posedge vclock)
      begin

            if (reset) counter <= 0;

            if ( state == ONE_PLAYER_GAME ) begin

                        phsync <= t_hsync;
                        pvsync <= t_vsync;
                        pblank <= t_blank;

                        name1_cstring <= (id == 2'b01) ?
80'h4A617669657220202020 : (id == 2'b10) ? 80'h436861726C6965202020: 80'hx;
                        score1_cstring <= p1_score >=0 && p1_score <10 ?
{8'd48,8'd48+p1_score,32'h30303030}
                                                  : p1_score >=10 && p1_score
<19 ? {8'd49,8'd48+p1_score-10,32'h30303030}
                                                  : p1_score >=20 && p1_score
<29 ? {8'd50,8'd48+p1_score-20,32'h30303030}
```

```
                                                  : p1_score >=30 && p1_score
<39 ? {8'd51,8'd48+p1_score-30,32'h30303030} : 48'h303030303030;



                        pixel <= (name1_pixel== 7) ? name1_pixel :
(score1_pixel==7) ? score1_pixel
                                    : (player1_pixel > 3'd0) ? player1_pixel:
(enemy1_pixel > 3'd0) ? enemy1_pixel
                                    : (enemy2_pixel > 3'd0) ? enemy2_pixel :
(e1_bullet_pixel > 3'd0) ? e1_bullet_pixel
                                    : (e2_bullet_pixel > 3'd0) ?
e2_bullet_pixel : p1_bullet_pixel;




            end

            else if (state == START) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h20535041434520464F5243452044554F20;
                  message_cstring <=
240'h2055736520796F757220524649442074616720746F206C6F6720696E2020;

                  pixel <= (start_pixel== 7) ? 3'b001: message_pixel;


            end
            else if (state == LOGIN) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h20535041434520464F5243452044554F20;
                  input_cstring <=
176'h203120506C617965723A2050726573732022312222020;
                  input2_cstring <=
184'h203220506C61796572733A2050726573732022322222020;
                  pixel <= (start_pixel== 7) ? 3'b001: (input_pixel==7) ?
input_pixel: input2_pixel;



            end
            else if (state == ONE_PLAYER) begin

                  phsync <= t_hsync;
```

```
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h2020202020526561647 93F202020202020;
                  input_cstring <=
176'h202020202050726573732 0456E746572202020202020;


                  //name of user
                  name1_cstring <= (id[1:0] == 2'b01) ? 80'h4A617669657 22020
: (id == 2'b10) ? 80'h436861726C696520: 80'hx;
                  score1_cstring <= p1_score >=0 && p1_score <10 ?
{8'd48,8'd48+p1_score,32'h30303030}
                                                  : p1_score >=10 && p1_score
<19 ? {8'd49,8'd48+p1_score-10,32'h30303030}
                                                  : p1_score >=20 && p1_score
<29 ? {8'd50,8'd48+p1_score-20,32'h30303030}
                                                  : p1_score >=30 && p1_score
<39 ? {8'd51,8'd48+p1_score-30,32'h30303030} : 48'h303030303030;

                  pixel <= (start_pixel== 7) ? 3'b010: (name1_pixel== 7) ?
name1_pixel : (score1_pixel==7) ? score1_pixel: input_pixel;



            end
            else if (state == TWO_PLAYERS) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h2020202020526561647 93F202020202020;
                  input_cstring <=
176'h202020202050726573732 0456E746572202020202020;

                  pixel <= (start_pixel== 7) ? 3'b010: input_pixel;

            end
            else if (state == HOST_READY_WAITING) begin

                  counter <= (counter ==64800000) ? 0 : counter+1;

                  if (counter ==64799999)
                        num_of_dots <= num_of_dots+1;
                  else
                        num_of_dots <= num_of_dots;

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h2020202020202020206973205265616479;
                  start2_cstring <= (id[1:0] == 2'b01) ?
80'h4A6176696572202020 : (id == 2'b10) ? 80'h436861726C69652020: 80'hx;

                  if (id_packet_in[1:0] == 2'b01) begin
```

```
                        if (num_of_dots==0) message_cstring <=
{96'h57616974696E6720666F7220, 64'h4A61766965722020,
80'h746F206A6F696E202020};
                        else if (num_of_dots == 1) message_cstring <=
{96'h57616974696E6720666F7220, 64'h4A61766965722020,
80'h746F206A6F696E2E2020};
                        else if (num_of_dots == 2) message_cstring <=
{96'h57616974696E6720666F7220, 64'h4A61766965722020,
80'h746F206A6F696E2E2E20};
                        else message_cstring <=
{96'h57616974696E6720666F7220, 64'h4A61766965722020,
80'h746F206A6F696E2E2E2E};
                  end
                  else if (id_packet_in[1:0] == 2'b10 ) begin
                        if (num_of_dots==0) message_cstring <=
{96'h57616974696E6720666F7220, 64'h436861726C69652020,
80'h746F206A6F696E202020};
                        else if (num_of_dots == 1) message_cstring <=
{96'h57616974696E6720666F7220, 64'h436861726C69652020,
80'h746F206A6F696E2E2020};
                        else if (num_of_dots == 2) message_cstring <=
{96'h57616974696E6720666F7220, 64'h436861726C69652020,
80'h746F206A6F696E2E2E20};
                        else message_cstring <=
{96'h57616974696E6720666F7220, 64'h436861726C69652020,
80'h746F206A6F696E2E2E2E};
                  end
                  else message_cstring <= {96'h57616974696E6720666F7220,
64'h506C617965723220, 80'h746F206A6F696E2E2E2E};

                  pixel <= (start2_pixel== 7) ? 3'b011 : (start_pixel== 7) ?
3'b001: message_pixel;

            end
            else if (state == HOST_READY) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h20202020202020202020697320526561647479;
                  start2_cstring <= (id_packet_in[1:0] == 2'b01) ?
80'h4A617669657220202020 : (id_packet_in[1:0] == 2'b10) ?
80'h436861726C6965202020: 80'hx;


                  input_cstring <=
176'h202020204172652079 6F752052454144593F20202020;
                  input2_cstring <=
184'h20202020202050726573732045 6E746572202020202020;

                  pixel <= (input2_pixel== 7) ? input2_pixel: (input_pixel==
7) ? 3'b010: (start2_pixel== 7) ? 3'b011: (start_pixel== 7) ? 3'b001: 0;

            end
            else if ( state == TWO_PLAYER_GAME_HOST ) begin
```

```
                        phsync <= t_hsync;
                        pvsync <= t_vsync;
                        pblank <= t_blank;

                        name1_cstring <= (id[1:0] == 2'b01) ?
80'h4A617669657220303120 : (id[1:0] == 2'b10) ? 80'h436861726C6965203131:
80'hx;
                        score1_cstring <= p1_score >=0 && p1_score <10 ?
{8'd48,8'd48+p1_score,32'h30303030}
                                                 : p1_score >=10 && p1_score
<19 ? {8'd49,8'd48+p1_score-10,32'h30303030}
                                                 : p1_score >=20 && p1_score
<29 ? {8'd50,8'd48+p1_score-20,32'h30303030}
                                                 : p1_score >=30 && p1_score
<39 ? {8'd51,8'd48+p1_score-30,32'h30303030} : 48'h303030303030;

                        name2_cstring <= (id_packet_in[1:0] == 2'b01) ?
80'h4A617669657220303120 : (id_packet_in[1:0] == 2'b10) ?
80'h436861726C6965203131: 80'hx;
                        score2_cstring <= p2_score >=0 && p2_score <10 ?
{8'd48,8'd48+p2_score,32'h30303030}
                                                 : p2_score >=10 && p2_score
<19 ? {8'd49,8'd48+p2_score-10,32'h30303030}
                                                 : p2_score >=20 && p2_score
<29 ? {8'd50,8'd48+p2_score-20,32'h30303030}
                                                 : p2_score >=30 && p2_score
<39 ? {8'd51,8'd48+p2_score-30,32'h30303030} : 48'h303030303030;

                        pixel <= (name1_pixel== 7) ? name1_pixel :
(score1_pixel==7) ? score1_pixel
                               : (name2_pixel== 7) ? name2_pixel :
(score2_pixel==7) ? score2_pixel
                               : (p1_bullet_pixel > 3'd0) ? p1_bullet_pixel :
(p2_bullet_pixel > 3'd0) ? p2_bullet_pixel
                               : (enemy1_pixel > 3'd0) ? enemy1_pixel :
(enemy2_pixel > 3'd0) ? enemy2_pixel
                               : (e1_bullet_pixel > 3'd0) ? e1_bullet_pixel :
(e2_bullet_pixel > 3'd0) ? e2_bullet_pixel
                               : (enemy3_pixel > 3'd0) ? enemy3_pixel :
(e3_bullet_pixel > 3'd0) ? e3_bullet_pixel
                               : (player1_pixel > 3'd0) ? player1_pixel :
player2_pixel;

            end
            else if ( state == TWO_PLAYER_GAME_CLIENT ) begin

                        phsync <= t_hsync;
                        pvsync <= t_vsync;
                        pblank <= t_blank;

                        name1_cstring <= (id_packet_in[1:0] == 2'b01) ?
80'h4A617669657220303120 : (id_packet_in == 2'b10) ?
80'h436861726C6965203131: 80'hx;
                        score1_cstring <= p1_score_in >=0 && p1_score_in <10
? {8'd48,8'd48+p1_score_in,32'h30303030}
                                                 : p1_score_in >=10 &&
p1_score_in <19 ? {8'd49,8'd48+p1_score_in-10,32'h30303030}
```

```
                                                        : p1_score_in >=20 &&
p1_score_in <29 ? {8'd50,8'd48+p1_score_in-20,32'h30303030}
                                                        : p1_score_in >=30 &&
p1_score_in <39 ? {8'd51,8'd48+p1_score_in-30,32'h30303030} :
48'h303030303030;

                        name2_cstring <= (id[1:0] == 2'b01) ?
80'h4A617669657220303120 : (id[1:0] == 2'b10) ? 80'h436861726C6965203131:
80'hx;
                        score2_cstring <= p2_score_in >=0 && p2_score_in <10
? {8'd48,8'd48+p2_score_in,32'h30303030}
                                                        : p2_score_in >=10 &&
p2_score_in <19 ? {8'd49,8'd48+p2_score_in-10,32'h30303030}
                                                        : p2_score_in >=20 &&
p2_score_in <29 ? {8'd50,8'd48+p2_score_in-20,32'h30303030}
                                                        : p2_score_in >=30 &&
p2_score_in <39 ? {8'd51,8'd48+p2_score_in-30,32'h30303030} :
48'h303030303030;

                        pixel <= (name1_pixel== 7) ? name1_pixel :
(score1_pixel==7) ? score1_pixel
                               : (name2_pixel== 7) ? name2_pixel :
(score2_pixel==7) ? score2_pixel
                               : (p1_bullet_pixel > 3'd0) ? p1_bullet_pixel :
(p2_bullet_pixel > 3'd0) ? p2_bullet_pixel
                               : (enemy1_pixel > 3'd0) ? enemy1_pixel :
(enemy2_pixel > 3'd0) ? enemy2_pixel
                               : (e1_bullet_pixel > 3'd0) ? e1_bullet_pixel :
(e2_bullet_pixel > 3'd0) ? e2_bullet_pixel
                               : (enemy3_pixel > 3'd0) ? enemy3_pixel :
(e3_bullet_pixel > 3'd0) ? e3_bullet_pixel
                               : (player1_pixel > 3'd0) ? player1_pixel :
player2_pixel;

            end
            else if (state == GAME_OVER) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h2020202047414D45204F56455220202020;
                  input_cstring <=
176'h507265737320456E74657220746F2052657374617274;

                  pixel <= (start_pixel== 7) ? 3'b100: input_pixel;

            end
            else if (state == YOU_WIN) begin

                  phsync <= t_hsync;
                  pvsync <= t_vsync;
                  pblank <= t_blank;

                  start_cstring <= 136'h202020596F752057696E21202020;
                  input_cstring <=
176'h507265737320456E74657220746F2052657374617274;
```

```
                        pixel <= (start_pixel== 7) ? 3'b110: input_pixel;

                end
                else begin

                        phsync <= t_hsync;
                        pvsync <= t_vsync;
                        pblank <= t_blank;

                        pixel<= 0;


                end
        end


endmodule
```

Ship Sprite Module

```
module ship_sprite
   #(parameter HEIGHT = 64,      // default height: 64 pixels
               WIDTH = 64)   // default color: white
   (input vclock,
       input [10:0] x,hcount,
    input [9:0] y,vcount,
       input display,
    output reg [2:0] pixel
       );
      wire [3:0] pixel_out;
      reg [11:0] addr;
      reg [9:0] ship_addr;
      reg [10:0] ship_hcount;
      reg [11:0] raddr;


      //ship36x54 ship(.addr(addr),.clk(vclock),.dout(dout));
      ship36x54 ship(addr,vclock,pixel_out);

      //sixteen_to_eight_colors
conversion(.vclock(vclock),.pixel_in(dout),.pixel_out(pixel_out));


      always @ ( posedge vclock ) begin

              if (~display) begin
                      pixel <= 0;
              end
              else begin

                      if ((hcount >= x && hcount < (x+WIDTH)) && (vcount >= y &&
vcount < (y+HEIGHT))) begin
                              addr <= raddr;
```

```
                        pixel <= (pixel_out ==0 || pixel_out ==7 ) ? 3'b000:
(pixel_out==15 || pixel_out==8) ? 3'b111
                          : (pixel_out ==12 || pixel_out ==4) ? 3'b001:
(pixel_out==10 || pixel_out==2) ? 3'b010
                          : (pixel_out ==6 || pixel_out ==14) ? 3'b011:
(pixel_out==1 || pixel_out==9 ) ? 3'b100
                          : (pixel_out ==13 || pixel_out ==5 ) ? 3'b101:
(pixel_out==11 || pixel_out==3) ? 3'b110: 3'b111;
                  end
                  else begin
                        addr <= 0;
                        pixel <= 0;
                  end
            end
      end

always @ (posedge vclock) begin
ship_addr <= (vcount-y); // how far into the picture
// vcount is vertically with respect to the top-left
// hand side of DK
ship_hcount <= (hcount-x); // how far into the picture
// hcount is width-wise with respect to the top-left
// hand side of the DK to be displayed
raddr <= {ship_addr[5:0], ship_hcount[5:0]}; // ship_hcount is 6 bits so the
// shift is equivalent to multiplication by 64
// saves significant computation time
end
endmodule
```

Bullet Sprite Module

```
module circle
   #(parameter RADIUS = 32,      // default height: 64 pixels
               COLOR = 3'b111)   // default color: white
   (input vclock,
       input [10:0] x,hcount,
    input [9:0] y,vcount,
       input display,
    output reg [2:0] pixel);

      reg[21:0] x_reg;
      reg[20:0] y_reg;

      reg [2:0] pixel_1;

   always @ (posedge vclock) pixel <=pixel_1;

      always @ ( posedge vclock ) begin

            if (~display) begin
                  pixel_1 <= 0;
            end
            else begin
```

```
                    if (hcount > x) x_reg <= (hcount-x)*(hcount-x);

                    else x_reg <= (x-hcount)*(x-hcount);

                    if (vcount > y) y_reg <= (vcount-y)*(vcount-y);

                    else y_reg <= (y-vcount)*(y-vcount);

                    if ( (x_reg + y_reg) < (RADIUS*RADIUS) )
                          pixel_1 <= COLOR;
                    else
                          pixel_1 <= 0;
              end
  end
  endmodule
```