ECE 540/558 Final Project Report

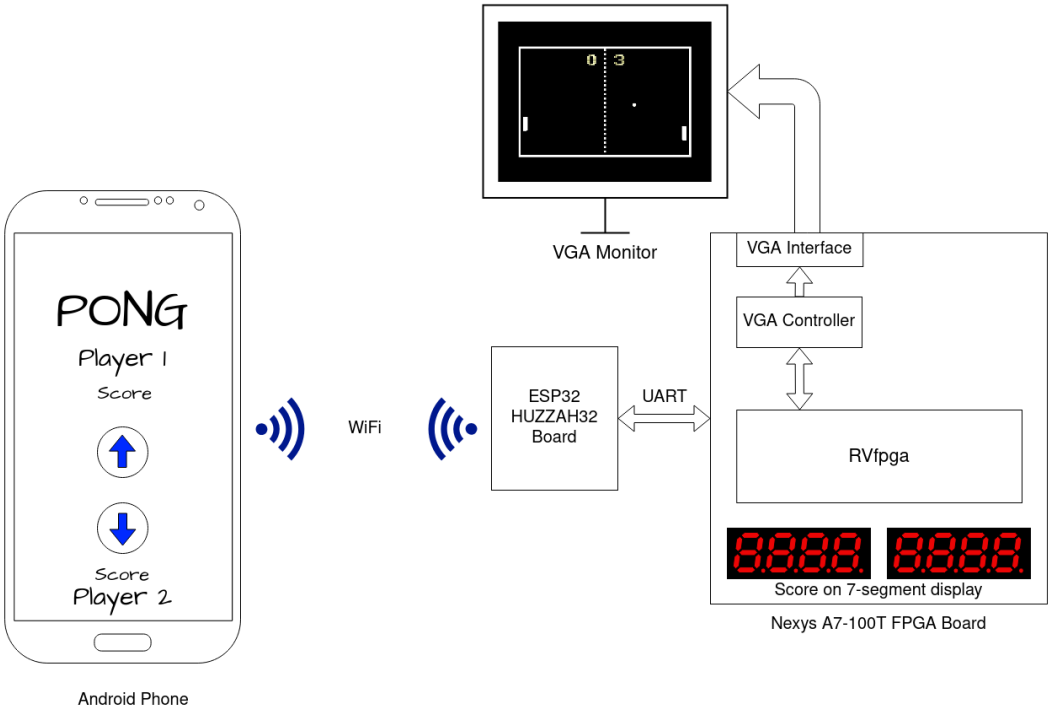
Arcade Game: Pong

Team Members: Atharva Lele, Ayush Srivastava, Mehul Shah
Github: https://github.com/atharvalele/ece540_ece558_pong

Project Description

An arcade game of pong running on the FPGA. Pong paddles are controlled by an Android phone interfaced over a wireless network.

Block Diagram

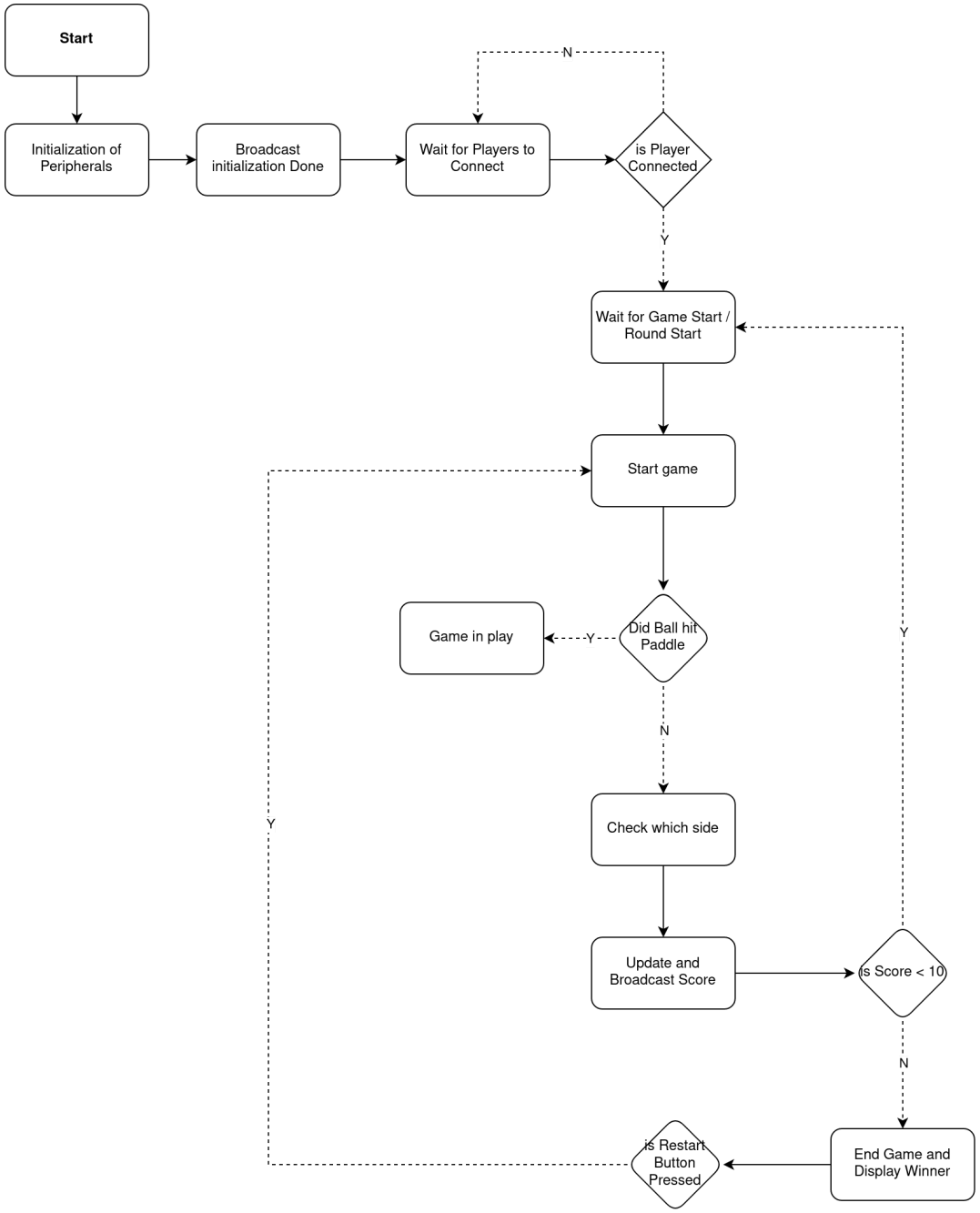


Theory of Operation

RVfpga

The RVfpga softcore implemented on the Nexys 4 board is the brains of the entire operation. It runs the game logic for the project, while also driving the display for the game. It interacts with the outside world via the ESP32 connected to it over UART.

The operation flow can be observed in the diagram shown below



The UART has been configured to work at a baud rate of 115200 in 8N1 mode.

The controllers (Android) communicate over WiFi with the ESP32, which passes on the data to RVfpga over UART. To communicate between the ESP32 and RVfpga we designed a simple communication protocol consisting of small strings delimited by commas. This allowed us to control the state changes in the game logic and distinguish between the commands sent by the player.

```
// Read data from the UART FIFO
sw_fifo_read(&uart_rx_fifo, message, rx_index);
while (i < rx_index) {
              // Game start message: G,1
              if ((message[i] == 'G') && (message[i + 1] == ',') && (message[i + 2] ==     '1')) {
              // Start game
              pong_set_state(PONG_GAME_START);
              i += 3;
              }
}
```

The code above shows the parse logic for one command: Game Start.

The game logic also employs the use of PTC (PWM, Timer, Counter). The PTC is connected to SW IRQ #3. It generates an interrupt every 1 ms or “tick”.

```
/* Timer ISR */
void timer_isr(void)
{
        /* Re-init with needed values */
        timer_init();
        /* Stop the generation of the specific external interrupt */
        bspClearExtInterrupt(3);
        /* UART Timeout */
        uart_timeout();
        /* Count ticks */
        msec++;
        delay_cnt++;
        /* Render game every GAME_RENDER_MS milliseconds */
        if (pong_started)
        if (msec % GAME_RENDER_MS == 0)
                pong_render = 1;
        if (msec >= 1000) {
        msec = 0;
        secflag = 1;
        }
}
```

The above code shows the Timer ISR, which is responsible for enabling the flag which tells the program to render the game, as well as counting down the UART timer.

The project being an arcade game required the implementation of a Display driver so that the game could be rendered through the onboard VGA connector. We designed a VGA controller with a Dual Port Block RAM (Vivado Memory Generator IP). The Dual Port RAM allowed us to simultaneously read from and write to locations on the block RAM.

```
        // Block RAM module instance
        blk_mem_gen_0 vga_image_mem (
        .clka(wb_clk),    // input wire clka
        .wea(wea),        // input wire [0 : 0] wea
        .addra(addra),  // input wire [18 : 0] addra
        .dina(dina),      // input wire [3 : 0] dina
        .clkb(clock),     // input wire clkb
        .addrb(pixel_num_wire),  // input wire [18 : 0] addrb
        .doutb(vga_data)  // output wire [3 : 0] doutb
        );
```

Port A of the Block RAM was solely meant for the purpose of writing and Port B was used to constantly read from it. We used the Display Timing Generator from Project 3 to generate the pixel addresses for the data that needs to be displayed. Port A was directly connected to the wishbone bus which allowed us to assign it a memory space on the wishbone multiplexer and used it as a memory-mapped peripheral. It occupies the address space 0x80003000. To add finer control and additional features we exposed a few registers of the peripheral.

0x00 used to enable Write Enable for Port A
0x04 used to write the address where data needs to be written
0x08 used to send data that is to be written
0x0C used to draw a horizontal line left to right
0x14 used to draw a vertical line top to bottom
0x10 used to draw a horizontal line right to left
0x18 used to draw a vertical line from bottom to top
The last two addresses weren't used for the project however were still implemented.

```
case (i_wb_adr)
                // 0x80003000: mapped to BRAM Write Enable
                0: begin
                        if (i_wb_sel[0]) wea <= i_wb_dat[0];
                end
                // 0x80003004: mapped to BRAM Address A
                4: begin
                        if (i_wb_sel[0]) addra[7:0] <= i_wb_dat[7:0];
                        if (i_wb_sel[1]) addra[15:8] <= i_wb_dat[15:8];
                        if (i_wb_sel[2]) addra[18:16] <= i_wb_dat[18:16];
                end
                // 0x80003008: mapped to BRAM Data In A
                8: begin
                        if (i_wb_sel[0]) dina[3:0] <= i_wb_dat[3:0];
                end
                // 0x8000300C: horizontal line, L to R, increment address by 1
                12: begin
                        if (i_wb_sel[0]) dina[3:0] <= i_wb_dat[3:0];
                        addra <= addra + 1;
                end
                // 0x80003010: horizontal line, R to L, decrement address by 1
                16: begin
                        if (i_wb_sel[0]) dina[3:0] <= i_wb_dat[3:0];
                        addra <= addra - 1;
                end
                // 0x80003014: vertical line, top to bottom, increment address by 640
                20: begin
                        if (i_wb_sel[0])  dina[3:0] <= i_wb_dat[3:0];
                        addra <= addra + 640;
                end
                // 0x80003018: vertical line, bottom to top, decrement address by 640
                24: begin
                        if (i_wb_sel[0]) dina[3:0] <= i_wb_dat[3:0];
                        addra <= addra - 640;
                end
                endcase
```

On completion of the Verilog implementation, we designed a display driver to do the basic tasks like drawing a pixel. This was necessary as we aimed to use a hardware-agnostic graphics library to draw various shapes and patterns on the display.

```
void draw_pixel(u16_t x, u16_t y, u08_t value)
{
        u32_t addr = calculate_frame_addr(x, y);
        WRITE_REG(VGA_BASE_ADDR, VGA_PIXEL_ADR_OFFSET, addr);
        WRITE_REG(VGA_BASE_ADDR, VGA_PIXEL_DAT_OFFSET, value);
}
```

Using the additional functions we created in the address space we were able to speed up the drawing process by providing functions like drawing horizontal and vertical lines.

```
/* Draw horizontal line */
void display_draw_hline(s16_t x0, s16_t y0, u16_t width, u08_t val)
{
	volatile u16_t w = width - 1;

	draw_pixel(x0, y0, val);

	while (w--) {
	draw_pixel_auto_inc_hline(val);
	}
}

/* Draw vertical line */
void display_draw_vline(s16_t x0, s16_t y0, u16_t height, u08_t val)
{
	volatile u16_t h = height - 1;
	draw_pixel(x0, y0, val);

	while (h--) {
	draw_pixel_auto_inc_vline(val);
	}
}
```

ESP32

The ESP32 was an integral part of the communication in the project. It was responsible for routing all messages from the android devices to the RVfpga and back while also maintaining data regarding the players like the assigned IP address and name of the player.
The ESP32 on boot would set up a Wireless Access Point for the android devices to connect. On establishing a connection, the user would open the app for the game and enter their name. The first player to send the name was assigned the status of player one, and the device's IP address would get stored in the structure associated with the name sent by the player.

```
struct player_t
{
	IPAddress ip;
	String name;
};
```

Once the second player connected and sent the name it would be using, the ESP would communicate to the RVfpga that the users have connected, triggering a state change in the RVfpga. The game was now ready to be started. At the start of the game, Player 1 had the control to start the game, and as soon as the player moved their paddle, the ESP would send a Game Start and Round Start message in tandem (Message pattern as shown above). The android device would send continuous messages with the commands for the paddles. The ESP listens for these messages using a UDP listener set up at the start of execution. UDP uses port 2000 to transmit & 2001 to receive on the ESP and vice versa on the android devices.

```
void udpListener() {
	if (udp.listen(UDP_RX_PORT)) {
	udp.onPacket([](AsyncUDPPacket packet) {
		packet.setTimeout(0);
		parsePacket(packet);
	});
```

```
        }
}
```

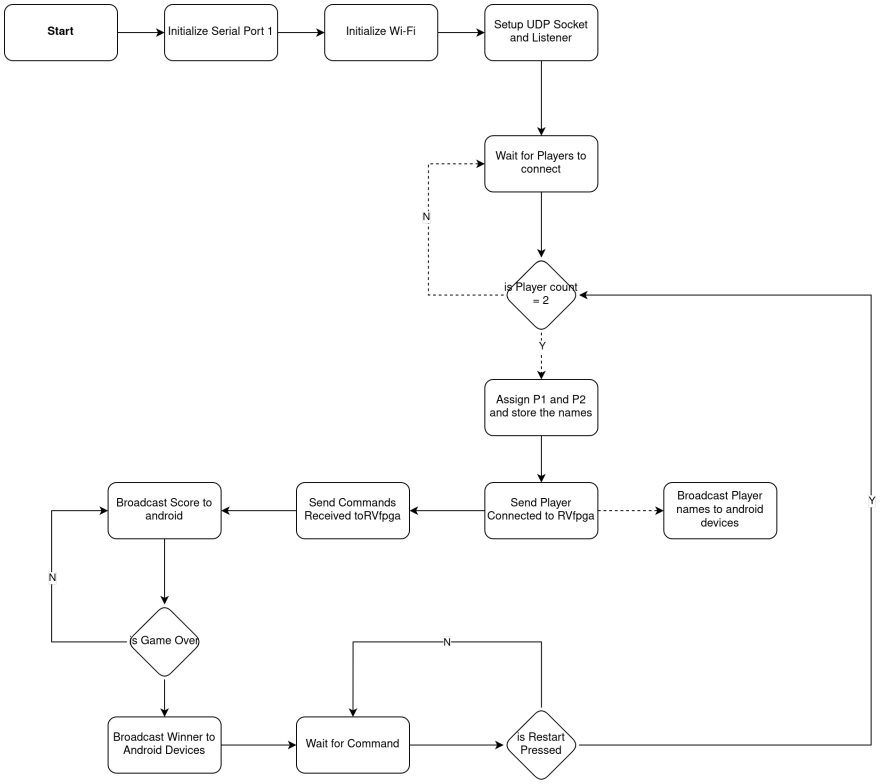
The UDP listener would invoke parse packet passing the UDP packet as an argument and decode the message sent by the android device for the ESP. Based on the message, the ESP either passed on the message or slightly modified it before sending it through based on the message.

```
        if (packet.remoteIP() == p1.ip) {
                //Serial.println(packet.length());
                if ((game_on == false) && (round_on == false)) {
                sendGameStart();
                game_on = true;
                }

                if ((round_on == false)) {
                sendRoundStart();
                round_on = true;
                }
                sendMessagePong(P1, messageBuffer);
                // @TODO: Send message over serial to rvfpga
        }


void sendMessagePong (int player, String packet) {
        String messageString;
        // Construct Message to Send to NEXYS
        switch (player)
        {
        case P1:
                messageString = PLAYER + String(",") + String(P1) + "," + packet;
                Serial.println(messageString);
                Serial1.println(messageString);
                break;
 // Removed code to simplify demonstration
        }
}
```

The above example shows one such scenario where the message is modified.

For the most part, all messages from the RVfpga (mainly scores of the players) get passed onto both Android devices except for messages that trigger haptic feedback on the android device. These were targeted based on which paddle touched the ball.

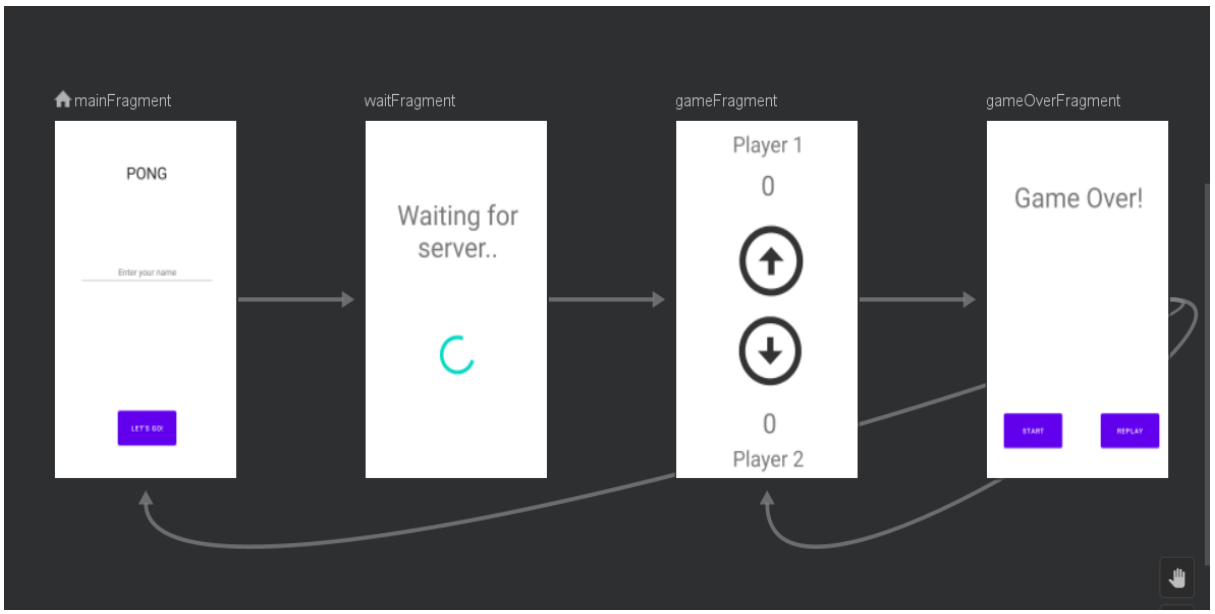
The diagram below shows the general flow of execution on the ESP32 end.



Android

The implementation of the android application is fairly simple. It consists of four fragments and can be seen in the diagram below:

- Main Fragment
- Wait Fragment
- Game Fragment
- Game Over Fragment



The Main fragment is the first fragment that shows up when the application is started. It consists of a text box which accepts the name of the player. This is then passed on to the Wait Fragment via the means of SafeArgs. The wait fragment uses the UDP send function to send the data over to the ESP32. On receiving a response from the ESP that the players have connected and the game is ready to be started the application transitions from the Wait Fragment to the Game Fragment. The Game Fragment facilitates the

control of the paddles in the game. Each player can hit the up or down button which sends a simple message (1 for down and 0 for up) over UDP. The score text boxes are constantly updated by messages received via UDP from the ESP32 to reflect the current score of the player once a round is finished. Once the game is over another transition occurs from the Game Fragment to the Game Over Fragment. This fragment displays the winner of the game and presents the player with two choices. They can either replay the game (Game Over Fragment back to the Game Fragment) or begin with new user names (Game Over Fragment to Main Fragment).

All UDP communications mentioned above have been implemented in the MainActivity. The sendUPDMessage is a function of the main activity and is called from the various fragments.

```
fun sendUDPMessage(message: String) {
        // Hack Prevent crash (sending should be done using an async task)
        val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
        StrictMode.setThreadPolicy(policy)
        try {
            // Open a port to send the package
            val socket = DatagramSocket()
            socket.broadcast = true
            // Get the message from function parameters and make it into a byte
array
            val sendData = message.toByteArray()
            // Craft the packet with the parameters
            val sendPacket = DatagramPacket(
                sendData,
                sendData.size,
                InetAddress.getByName(UDP_IP_ADDR),
                UDP_SEND_PORT
            )
            socket.send(sendPacket)
        } catch (e: IOException) {
            Log.e("MainActivity", "IOException: " + e.message)
        }
    }
```

The receive function is implemented in its own thread and calls the `parsePacket()` function to trigger various transitions and events in the application.

Individual Contributions

Atharva Lele

- Board bring-up of RVfpga
    - Setting up the UART and Timer
- Implemented the UART interrupt and software FIFO
- Implementation of the game logic
- Added the extra primitive functions on the VGA controller that adds the ability to draw lines
- UDP Receive thread implementation

Ayush Srivastava

- Bring up the VGA controller
- Interfacing the BlockRAM with the wishbone bus
- Design the driver for the VGA controller adding the ability to draw a pixel on the screen
- Design and implementation of the communication between ESP, Android, and RVfpga
- Implementation of the BLE Stack on ESP32

Mehul Shah

- Design of the Android Application fragments
- Implementation of the UDP functions to send and receive data
- Development of the parser for the communication protocol
- Implementation of the Navigation Graph to navigate between fragments and passing data via Safe Args

Goals we set for ourselves.

Committed Deliverables

ECE540: FPGA side implementation

- Running the RVfpga softcore
- Implementation of Pong game
- The game screen displayed via VGA display
- Scores displayed on the 7-segment display
- ESP32 for data transfer over Wi-Fi (using UDP)

ECE558: Android app implementation

- Display scoreboard on the Android Application
- Android Application for controlling paddles with buttons
- Connect to FPGA using Wi-Fi (using UDP)

The Stretch Goals we achieved

ECE558: Android app implementation

- Haptic feedback when the ball strikes the paddle

Stretch Goals we couldn't achieve

- Use of accelerometer on the Android device
- Audio Output
- Save game history
- Paddle size selection