ES-4 Final Project Writeup

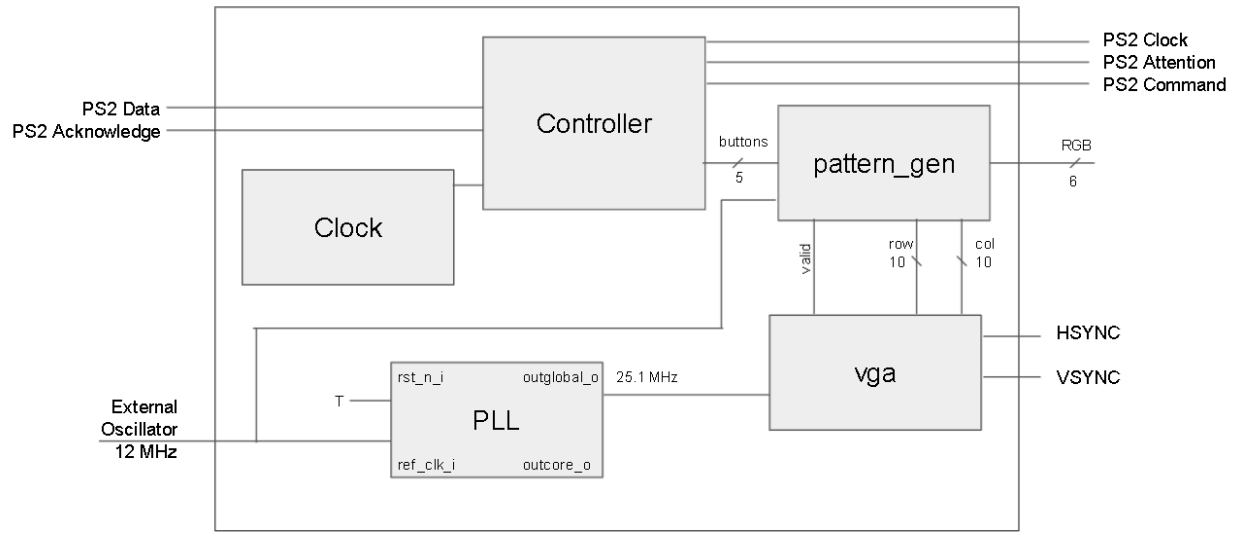
May 11, 2021

Professor Steven Bell

Project Team: Andrew Butcher, Chloe Lam, Kenny Lin, William Yao

Overview

This project implements a game modeled after the classic game Frogger to be controlled with a Playstation 2 Dance Mat Controller (typically used with the game Dance Dance Revolution).



The two major components of this project, the controller and the game, only interact through a single std_logic_vector, containing the data for the 5 usable buttons. When the controller module passes the button data to the game, the game affects its state accordingly. The game allows for the movement of the “frog” through the use of the 4 directional buttons and for a reset signal using the START button.

Github Repo: https://github.com/klin303/ES4-Spring-2021-Final-Project

Technical Description

Controller:

The controller module utilizes the basic Playstation 2 controller protocol. This protocol utilizes 5 signal wires to communicate between the controller and the “console”: Clock, Command, Acknowledge, Attention, and Data. The Data and Acknowledge signals are sent from the controller to the system, while the Clock, Command, and Attention signals go the other way.

The Clock signal operates as such: when at rest, the Clock signal is high. When it begins to oscillate, it does so at a frequency of about 187 kHz for 8 cycles and then returns to high for a length of time. This sequence is referred to as a byte, due to the fact that it will allow for the other signals to send data synchronous with the Clock but also still be interpreted as bytes. Because this particular controller is a “Standard Digital Pad” (to be explained with the Data section), the Clock must run for 5 bytes before holding a high signal for an extended period.

The Command signal operates as such: during the first two bytes of the Clock signal, the Command signal must transmit the hexadecimal values 0x01 and 0x42 respectively. These, like all numeric signals in this module, must be transmitted LSB first. Without receiving these Command signals, the controller will not communicate properly with the “console”. On this wire, a binary 1 is, as expected, encoded as HIGH.

The Attention signal operates as such: before the first byte, the Attention signal must be driven low. After the 5th and final byte, the Attention signal must be driven high again until the next cycle.

The Acknowledge signal operates as such: after each byte, the Acknowledge signal will be driven low by the controller for one clock period. After the final byte, the controller does not need to drive the Acknowledge signal.

The Data signal is the primary method of communication from the controller to the “console”. For every controller variant using the Playstation 2 protocol, there are three bytes of “preamble” before any button data is transmitted. On the first byte, Data will remain high. On the second and third byte, Data will transmit a numeric value to represent which type of controller it is. The Dance Mat controller sends hexadecimal values 0x41 and 0x5A on the second and third bytes respectively, signifying that this is a “Standard Digital” controller, and, as such, will only have 2 bytes of button data. The remaining 2 bytes are for transmitting which buttons are depressed on the current cycle, represented by the Data signal being driven low. See the figure below for details on which byte goes where.

BYTE	CMND	DATA							
01	0x01	idle							
02	0x42	0x41							
03	idle	0x5A	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
04	idle	data	SLCT			STRT	UP	RGHT	DOWN	LEFT
05	idle	data	L2	R2	L1	R1	/\	O	X	|_|

Because the data is sent synchronously with the Clock signal, one method of storing the buttons that are pressed is to use a shift register, which is the way we selected. On the rising edge of the Clock signal, the current bit of data is shifted in. After all 5 bytes of the clock signal on a given sequence, the 5 bits of the shift register corresponding to the directional inputs and the start button will be copied into a standard logic vector called buttons (NOTE: because any bit in Buttons is considered on when HIGH, the NOT of the bits in the shift register is copied). This button data is then ready to be used by the game. We only copied the 5 bits that were needed for the frogger game, but all 10 possible button inputs from the controller exist in the shift register. Because L2, R2, L1, and R1 do not exist on the controller, they are considered unusable and will remain constant.

One problem that we faced during this phase of the project was trying to get the correct clock period. Originally, we had believed that it had to be exactly 500 kHz, but none of the clock divider configurations we could come up with would adequately produce this frequency from the 48 MHz signal we were provided. Once we realized that there was an acceptable range of frequencies, this became significantly easier and we were able to create a functioning clock signal.

Another problem that we faced was that of properly copying the relevant shift register data into the Buttons signal. Because the Clock signal remains high after the last bit, we cannot rely on it to signal when to copy the shift register's data. If we tried to copy the data before Clock was driven high at the final bit, then the button data would not be in the right place, and there would be garbage values from the preamble bytes. The solution that we came up with was to have the shift register be copied once the byte counter rolls over to byte 6. This ensured that the register was properly updated before being copied.

One major problem we also encountered was that of attempting to program a state machine for the Command signal. The initial plan was to create a state machine that would change what Command was to output depending on the current byte. We ran into problems when trying to change to later bytes, as the state machine would never change. After many attempts at debugging, our solution was to simply hard code the Command various signals:

```
command <= '1' when (PS_count = 0) and (byte_count = 0) else
           '0' when (byte_count = 0) else
           '0' when (PS_count = 0) and (byte_count = 1) else
           '1' when (PS_count = 1) and (byte_count = 1) else
           '0' when (PS_count < 6) and (byte_count = 1) else
           '1' when (PS_count = 6) and (byte_count = 1) else
           '0' when (PS_count = 7) and (byte_count = 1) else
           '0';
```

This is an admittedly messy solution, and not one that we would recommend; however, it is completely functional, and did solve the encountered problem. As such, it was left in the code.

Display:

The game utilizes a VGA connector connected to the FPGA to display the graphics. For our purposes, we connected the VGA adapter to an HDMI capture connector which allowed us to display the VGA output on a monitor using VLC Media Player. The graphics for the VGA were driven by two modules connected to a top level module for the game. The VGA.vhd file generated the HSYNC and VSYNC signals for the VGA adapter and the pattern_gen.vhd file generated the game objects. The pattern_gen module would generate a 6-bit RGB output for each pixel on the screen.

For our game, all the game logic and graphics were created in the pattern_gen module. We were able to accomplish this because the game we chose to implement was relatively simple. If the game were more complex, we would have chosen to break the logic and graphics generation into two separate modules.

To create the game, we broke it down into three distinct components: the overall game state control, the frog movement/display, and the car movement/display.

First, we divided the 640x480 VGA display window into 15 40-pixel wide columns and 12 40-pixel wide columns to simplify the movement of game pieces. We then decided to make the top and bottom rows safe areas for the frog to start and end.

To implement the frog movement, we took the 12 MHz frequency from the FPGA and divided it down to a roughly 5 Hz frequency. We did this so the frog would not move too fast across the screen. To store the frog's position, we used two unsigned signals that would represent the frog's location on the 15x12 board. We then programmed the frog to be displayed as a green square using the frog's indexed row and column position. Finally, we connected a Digital Discovery to the FPGA to test the frog's movement. Since this was the first part we implemented, we tried many different clock signals before finding one that would move the frog smoothly.

Next we implemented a simple state machine to represent what stage of the game the player was in. The table below illustrates the four different states, what they accomplish, and how the program moves between them.

State	**Pregame**	**New Level**	**Running**	**Over**
Purpose	Initial state at the beginning of the game which initializes the cars	Resets the frog to the start after a level is completed or the game starts	Runs when the player is in the process of playing a level	Displays the dead screen and waits for the player to restart the game
Next States	Moves to new level once the cars are initialized	Moves to running once the frog is reset and player isn't pressing forward	Either moves to new level if the player makes it across or over if the player runs into a car and dies	Waits for the player to hit the reset button then transitions to pregame to reset the game

Finally, we implemented the car movement and positioning. Unlike the frog, the cars would move across the screen left or right in 20 pixel increments whereas the frog moved across in 40 pixel increments. One of our major challenges was determining how to store the position of the cars and display them on the screen. We initially tried a 2D-array of std_logic, but encountered issues trying to traverse the array.

At first, we tried to use for and while loops to traverse the array, but we discovered that loops in VHDL function quite differently from loops in other languages. Therefore, we decided to change directions and represent the cars as an array of logic vectors. This allowed us to use concatenation on either end of the vector to move cars across the screen. We created another clock divider to get the cars moving at a reasonable speed. To display the cars, we used the column and row count provided by the VGA module and divided the values to check if a car was present at that pixel in the array. The image below shows how we moved the cars left and right.

```
temp_board(1) <= actual_board(1)(31) & actual_board(1)(0 to 30);
temp_board(2) <= actual_board(2)(1 to 31) & actual_board(2)(0);
temp_board(3) <= actual_board(3)(31) & actual_board(3)(0 to 30);
temp_board(4) <= actual_board(4)(1 to 31) & actual_board(4)(0);
temp_board(5) <= actual_board(5)(31) & actual_board(5)(0 to 30);
temp_board(6) <= actual_board(6)(1 to 31) & actual_board(6)(0);
temp_board(7) <= actual_board(7)(31) & actual_board(7)(0 to 30);
temp_board(8) <= actual_board(8)(1 to 31) & actual_board(9)(0);
temp_board(9) <= actual_board(9)(31) & actual_board(9)(0 to 30);
temp_board(10) <= actual_board(10)(1 to 31) & actual_board(10)(0);
```

Combining the Controller and Display:

Once both parts of the project were completed, we combined the code into a single project with a top level file managing inputs, outputs, and communication between the different modules.

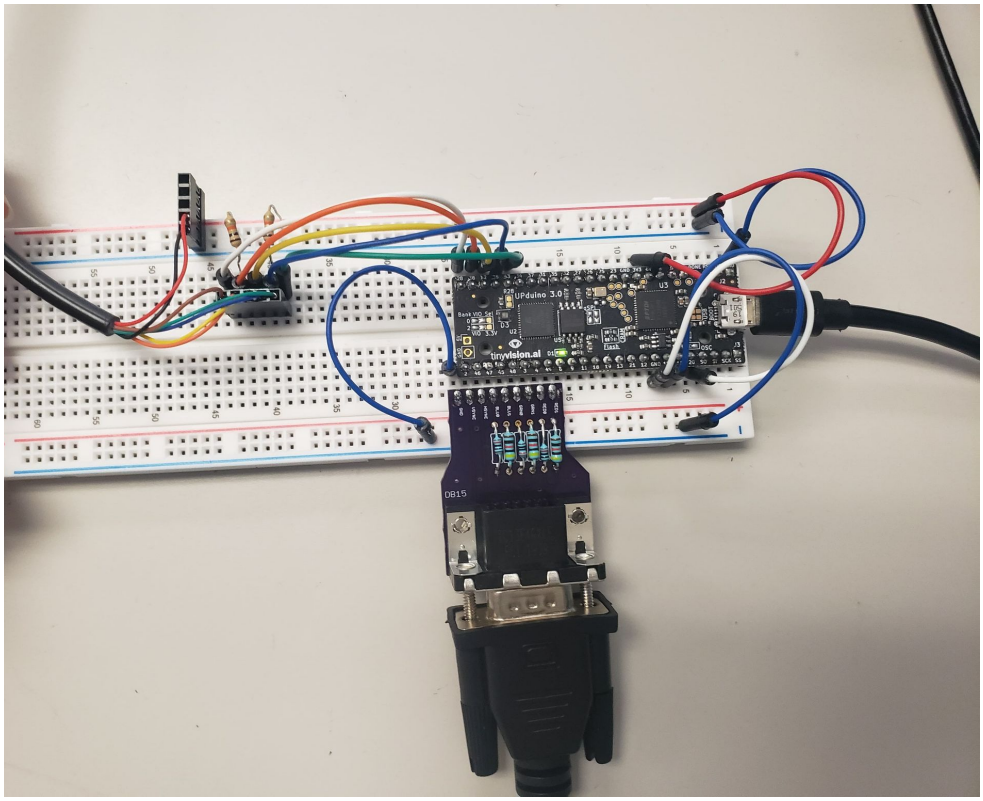
Results

Our project included a working display with a moving 40 by 40 green pixel frog with rows of moving 40 by 60 blue pixel cars which was displayed by the VGA connected to our FPGA. Some known bugs we noticed is that the screen tears as the cars are being drawn and moved. We assume that this is caused by a discrepancy between the clock and the car_clock as the car_clock is updating faster than the VGA clock, causing the VGA to display weirdly.

We also implemented a working Dance Dance Revolution Mat with the up, down, left, right, and start buttons. Something to experiment with could have been diagonal movement, however, we did not have the time to try this. If we had more time to work on Froggy Road, we would randomly generate the cars, learn how to store the cars in RAM, vary the length of cars (add buses and smart cars), and add levels that sped the cars up every time the player got to the end, as well as a menu screen.

Circuit:

Outputs:	Inputs:	Other:
Yellow - Attention	Green - Acknowledge	Red - Power
Orange - Command	Brown - Data	Black - Ground
Blue - Clock		



Video of project:
https://drive.google.com/file/d/1WBBh7wWOJzS-fuSovhYVGShfqijHuqe1/view?usp=sharing
https://drive.google.com/file/d/1fBdnTFzqYwe18X4WRJR3KnAahgoe2w13/view?usp=sharing
*Need to use Tufts gmail account.

Reflection

The project went relatively well as we were able to complete what we set out to do on time. We were able to get a “frog” to display and move through inputs from the dance mat (up, down, right, left) successfully. We were able to create and display “cars” that move left or right, depending on the row, which causes the user to lose on collision. We were also able to understand the PS2 protocol and get signals from the dance mat. What we would do differently

if we were to do this again would be to better plan out our modules and code as our code was not very modular. We would still split up in groups and integrate at the end.

Work Division

To complete the project on-time, we divided our group into two teams. Andrew and Chloe worked on interpreting input from the DDR mat while Kenny and Will created the game.

Controller: Andrew/Chloe

Counter Entity: Andrew

Controller Entity: Andrew and Chloe

Wiring: Chloe

Frogger Game: Kenny/Will

Frog Movement: Kenny

Game States: Kenny

Car Movement: Will

Debugging and Testing: Will and Kenny