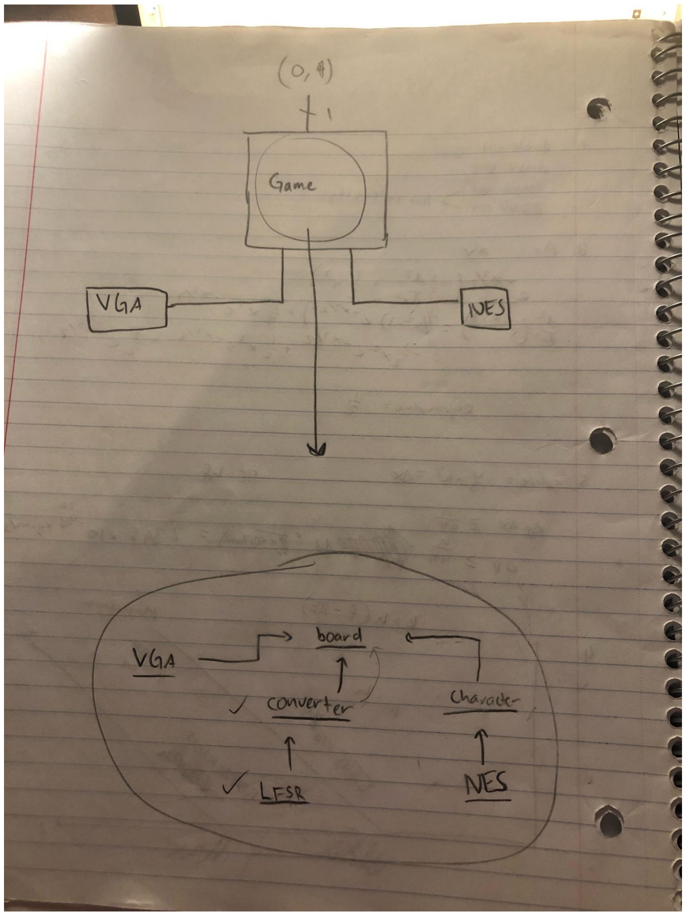
Code Repository:
https://github.com/lillianfullford/es4final

Overview

Our project consisted of a game where a player controller character positioned at the bottom of the screen is required to dodge falling debris while only capable of lateral movement. When an object does collide with the player, the game will end. The difficulty of the game gradually increases as time goes on, with this being accomplished by decrementing the accessed index of the counter connected to our FPGA's PLL output. This change in index results in the movements of the game becoming faster and faster, forcing the player to keep up or risk being hit by an object.

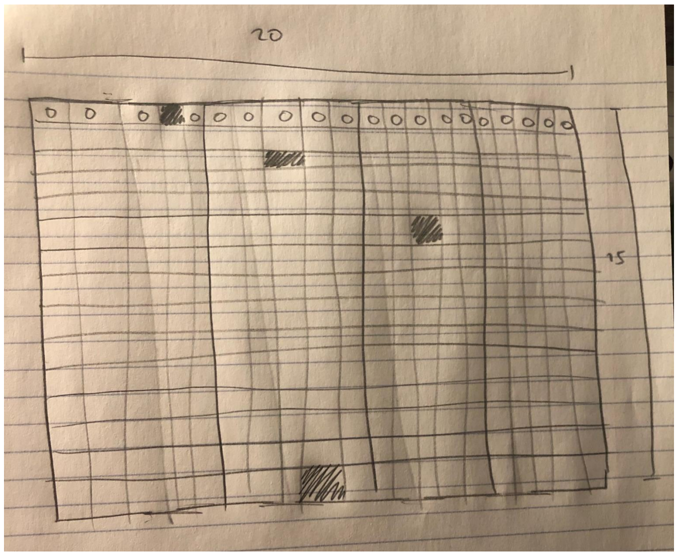
Technical Description

To build our game, we incorporated various components from the labs. The three labs that we completed to assist our project dealt with an NES controller for user input, programming memory, and programming a VGA driver to facilitate a display. Our final product used the memory and the VGA, but the VHDL for the NES controller input was removed at the last minute and replaced with two push buttons in order to simplify integration between the components. The way that all our files connected together is displayed below.



The memory section of our VHDL contained the portion that controls the functionality of our game. To build this, we had to break our game down into the basic components. As an overview, we knew our game fundamentally consisted of two moving components - our falling blocks, and our character. In order to simplify math and logic, we decided to represent the game board as a memory array. For the sake of simplicity, it was decided that our game objects would be displayed using black pixels, while the background would be white, allowing for the highest visual contrast.

Since the display format was 640 by 480 pixels, we decided to use a grid method to develop our system. Each block of pixels was a square 32 pixels across, creating a grid 20 blocks wide by 15 blocks tall. We decided that for each row, there would be a maximum of one falling block for the top 14 rows and that the bottom row would be entirely reserved for the character block's position, as illustrated in the image below.



The example shows a random frame in action. To create this, we decided to split everything up into various pieces. We decided to implement several separate files for the overarching memory. In order to create the behavior of the falling blocks, there was a single file that would randomly generate numbers (an LFSR), another that took these numbers and determined a block position using that information, a file that determined the characters position, and an overarching top module that combined all these parts and give an output result based on logic between the user inputs and the game's win conditions/function.

To represent the memory board, we decided to use a 15 entry array (14 downto 0). Each entry in this array represents an entire row of the board. Therefore, each row consisted of 20 bits. However, to create a possibility of having empty rows, we decided to make each row represented by 32 bits (5 downto 0). This allowed for the possibility of any randomly generated block to spawn off the display, making it impossible for the character to collide with it and therefore giving the appearance of having an empty row.

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

entity LFSR is
  port (
    clk : in std_logic;
    reset : in std_logic;
    count : out std_logic_vector(4 downto 0)
  );
end LFSR;

architecture synth of LFSR is

begin
  process (all) is
    begin
      if reset then
        count <= "00001";
      elsif rising_edge(clk) then
        count(0) <= count(1);
        count(1) <= count(2);
        count(2) <= count(3) xor count(3);
        count(3) <= count(4);
        count(4) <= count(0);
      end if;
  end process;
end;
```

To randomly generate each row (so that the blocks fell in different grid columns each time our board updates, we decided to use an LFSR. An LFSR randomly generates every value in a range in a random order. Since our rows were 32 bits wide, we decided that our LFSR would generate a number (5 downto 0) bar none, and the generated number would represent an index of a row in the memory board array. The LFSR code is shown to the left.

To standardize everything, and make everything fall from the top, we defined array index 0 as the top of the memory board. Therefore, the LFSR updated the top row of the memory board with every board update. Since the pixels were black if the falling object was there and white if the object was not, we decided to represent every object as a binary ‘1’ and every other empty space as a binary ‘0’. Therefore, since there was only one block in every row, there was one arbitrary index in each array entry to represent a block. The rest of the word was then filled out with 0s. This was accomplished using VHDL for loops, as shown in the code snippet to the right. The code created a 32 bit row that would then be inputted into the array.

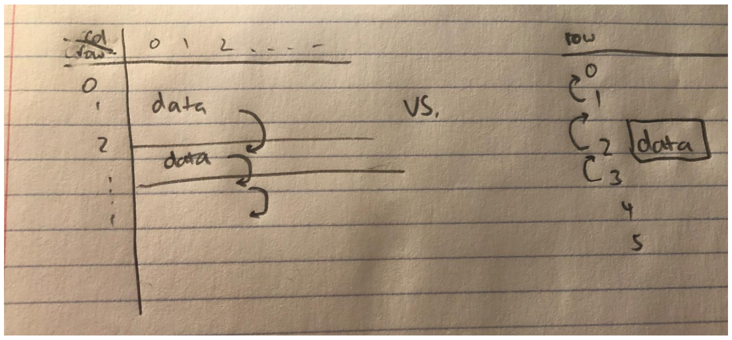
```
signal connector : std_logic_vector(4 downto 0);
signal output : unsigned (31 downto 0);
signal type_convert : unsigned(4 downto 0);
signal i : integer;

begin
  connection : LFSR port map (count=>connector);
  type_convert <= unsigned(connector);

  process
    begin
      for i in 0 to 31 loop
        if i = type_convert then
          output(i) <= '1';
        else
          output(i) <= '0';
        end if;

        wait for 5ns;
      end loop;

      row <= output;
  end process;
end;
```

With the code for the random blocks finished, the only other component that had to be imported into the top module was the code for the character. Like the rows that we generated for the blocks, the information for the character’s position was stored in a 32 bit word. However, this single word was updated with a shift of the ‘1’ bit along the word between the bounds. Therefore, the data in array(14) was never shifted up or down the array indices like the falling object data, but was instead replaced every time the board updated. The character file started out by taking two arbitrary switches as an input. It then assumed the starting position of the character at the start of the game was in the center of the board. This was at index 24, or halfway between indices 31 and 12, which are respective edges of the board. Therefore, this was either index 21 or 22. Then, the bounds were set so that the character would be unable to move past index 31 or

```
    arr(1) <= arr(0);
    arr(2) <= arr(1);
    arr(3) <= arr(2);
    arr(4) <= arr(3);
    arr(5) <= arr(4);
    arr(6) <= arr(5);
    arr(7) <= arr(6);
    arr(8) <= arr(7);
    arr(9) <= arr(8);
    arr(10) <= arr(9);
    arr(11) <= arr(10);
    arr(12) <= arr(11);
    arr(13) <= arr(12);
    arr(0) <= block_pos;
    arr(14) <= char_pos;
  end if;
```

index 12 so that it would always be in the displayed section of the board. Finally, we coded it so that if the right switch was pressed, the character's position index would decrease by one, and if the left switch was pressed, the character's position index would increase by one. Finally, the same for loop from the blocks was utilized to convert the index into a 32 bit word that represented the position of the player's character.

Initially, each index of the array was filled with words consisting of only 0s. However, each time the board updated, the data in each row would get shifted down one row. To make shifting easier, we set it so that instead of having the data move down the array (i.e. data in row 1 goes to row 2, row 2 goes to row 3), we had the array shift upwards around the data. This is illustrated in the image to the left. This allowed us to update the array efficiently.

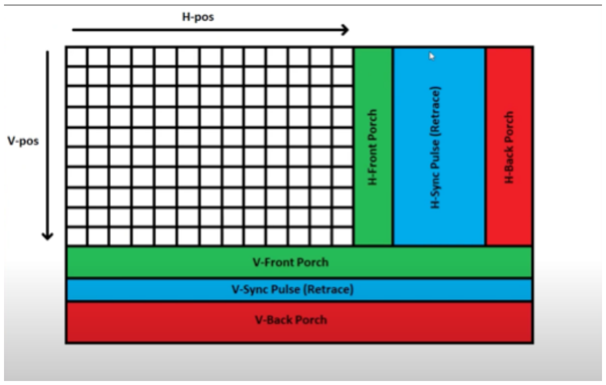


The top file assigned and updated the array continuously by taking the character file and the block files as inputs. The other input that it took was the VGA driver. The VGA gave a row and column index that would then be used to index the array inside the memory file. This would then return a value of "111" if the grid block was occupied by an entity (either the character or the blocks). Otherwise, it would return a value of "000". This was all done with a double for loop. After cycling through the entire board to determine what to display, the array would update. This entire cycle would continue until the end condition was met: if index 13 was ever equal to index 14, the entire board would be replaced with a white screen. This was done by looping through the array and replacing every index with a word full of 0s, which would fill the screen with white grid blocks.

```
if counter(4) = '1' then
    for int1 in 0 to 19 loop
        for int2 in 0 to 14 loop
            if int1 = vga_row and int2 = vga_col then
                if arr(int1)(int2) = '1' then
                    rgb <= "111";
                else
                    rgb <= "000";
                end if;
            end if;
        end loop;
    end loop;
```

```
        if arr(14) = arr(13) then
            for i in 0 to 14 loop
                arr(i) <= "00000000000000000000000000000000";
            end loop;
        end if;
    end process;
d:
```

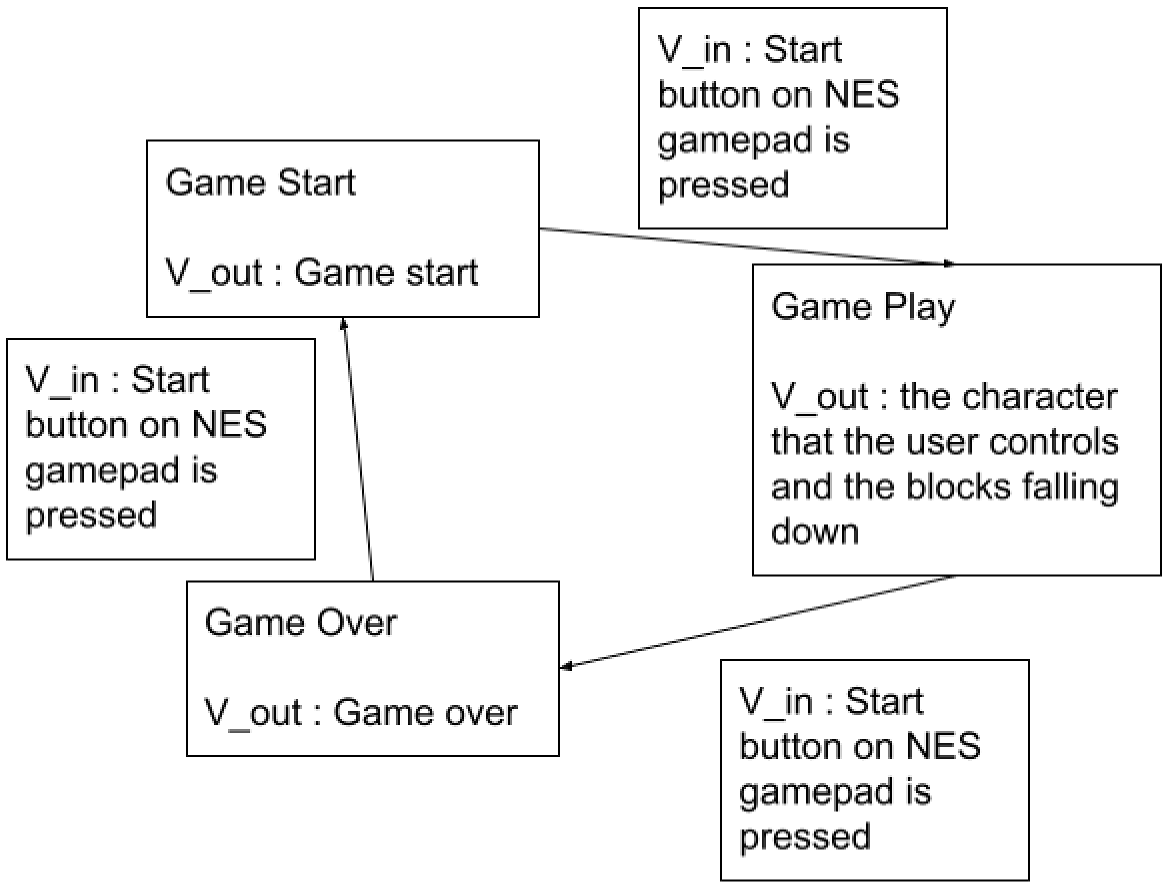
The VGA code essentially "read out" the position of the display and sent that information to the top module of the code. There were a few different factors that went into determining the position on the display. The diagram picture gives an in depth view of how the counters were set up in the VHDL code. The horizontal position was incremented by one pixel for a



complete row until the H-Front Porch was reached, then incremented vertically down by one pixel, then the back to the next horizontal row. The clock used was the PLL clock built into the FPGA. This clock was altered to the necessary 25.125 MHz output frequency. When the PLL did not generate correctly in Radiant, the entirety of the project was switched to using the open-source toolchain.

For the NES controller input, a latch pulse was sent from the FPGA to the controller at a frequency of approximately 50Hz, followed by seven pulses from the FPGA's clock signal. This latch signal would prompt the circuitry within the controller to respond with a sequence of pulses that corresponded to the state of the various buttons. These states were stored one at a time within a standard logic vector, replacing bits one through seven with bits zero through six, and then setting bit zero to the state of the button that was being read at the time. In order to avoid any invalid button detection, the collected information would only be passed on to the rest of the circuit once the entire 8 bit sequence was fully received from the controller. In an attempt to make integration with the other components of the project as easy as possible, the final outputs of the NES controller interpreter were simply std_logic's associated with each of the buttons, so that when a button was pressed, that button's wire would go high until the button was released.

The states that our game would have are a game start state, a game play state, and a game over state. Essentially we would have an extra button that allows us to transition from state to state. The states only change in one direction, and we did not decide on whether or not to make a reset button because we could essentially just unplug and replug the VGA and the game would be "reset" or it would more so be a hard reset.



V_in : Start button on NES gamepad is pressed
Game Start
V_out : Game start
Game Play
V_out : the character that the user controls and the blocks falling down
V_in : Start button on NES gamepad is pressed
Game Over
V_out : Game over
V_in : Start button on NES gamepad is pressed

Results

We have minimal results to show as a result of the VGA display failing to display leading up to the final project presentation. Hours were spent working with both Ryan (TA) and Professor Bell on possible fixes for the VGA issues, but nothing ended up working for the final presentation. Eventually, we attempted to restart from scratch and dig up as old of code as we could find to try to rebuild a VGA display that did not require the memory inputs, however this was unsuccessful for the short time frame that we had. Given more time, we would have completed a "recode" of a hardcoded VGA display. Then, we would have readded the memory components. Because over the course of the break, each component worked individually, it can be assumed that given the proper restructuring of the top level file, the components should work together. This applies to both the NES controller and the memory. These components were also checked over by Michael and Ryan. If the display and the memory talked to each other correctly and a display was visible, we would have been able to confirm that each component worked.

The results that we would have if everything was working properly is the same as the overview where we would have a character that we can move side to side with blocks falling from the sky randomly. The display would show the background being completely white pixels and every block and the character being black symbols. The character would essentially also just be a black block, but if everything was working we would try to change the color of the character block to something that is not black or white.

One of the largest issues we encountered was faulty hardware. The VGA hardware ended up being faulty and resulted in last minute rewiring of the VGA breakout board. Lily attempted to rewire the VGA adapter and Tom ended up resoldering another VGA connector board the night the project was due after multiple “once overs” of code that made completely logical sense. Additionally, it seemed that a larger problem with the VGA display was the adapter that was used seemed to be faulty. We ended up wondering if the mixing of VGA adapters and cords caused unforeseen issues in the display. After various consultations with other classmates and TA’s, we were unable to figure out what was wrong with either the code or the adapters, until a fellow student discovered a display issue with an adapter that we had been using to debug the display and loaned us one of his. The display had previously just been a black screen and then turned to a rainbow color. We didn’t really think too much about it, since we thought it was because our code was just malfunctioning horribly, but if it’s due to the VGA adapter then it would make sense why no matter what help we received nothing functioned properly.

Additionally, the NES controller was sat on and unfortunately broke the night before our final presentation. As a result Lily sat in Nolop resoldering and crimping the NES controller until she was at least somewhat satisfied that it would work again.

Reflection

The parts that went well was the construction of the NES gamepad and supposedly the memory portion of the game, but we were unable to check this part. The VGA did not go well in the sense that we just did not know what was wrong, since it could have been something wrong with the hardware and with our code. The issue with our code in the end is very unlikely because of Professor Bell and Ryan helping us so much. So there is a high chance that the error was due to hardware. We should have tested our circuitry with different hardware pieces just to make sure that it was working, instead of blantletly assuming that everything was wrong with our code.

If we had to do it again, we would first start with the building of the VGA as a group instead of splitting the work up and starting that way since the VGA is the most important part of the project because that’s where all of our visuals come from. The construction of the project was poorly planned and we probably should have attempted for all of us to work together for every part so that we all know exactly what’s going on with each segment of our project. There was the issue of all of us just not having enough time to work on the project because of how many other things were going on at the same time, but that is out of our control. Then finally we should first check that all of our hardware is working perfectly fine before assuming that they all work perfectly because if the hardware doesn’t work then nothing is going to work. We’re not too sure on how to check if the hardware is working properly besides to check if a voltage is going through each wire of it or not, so we would need to figure this out.

Work Division

Edwin and Tianyi did the memory vhdl files, Tom did the NES gamepad, and Lily did the VGA. We all worked on the final report.