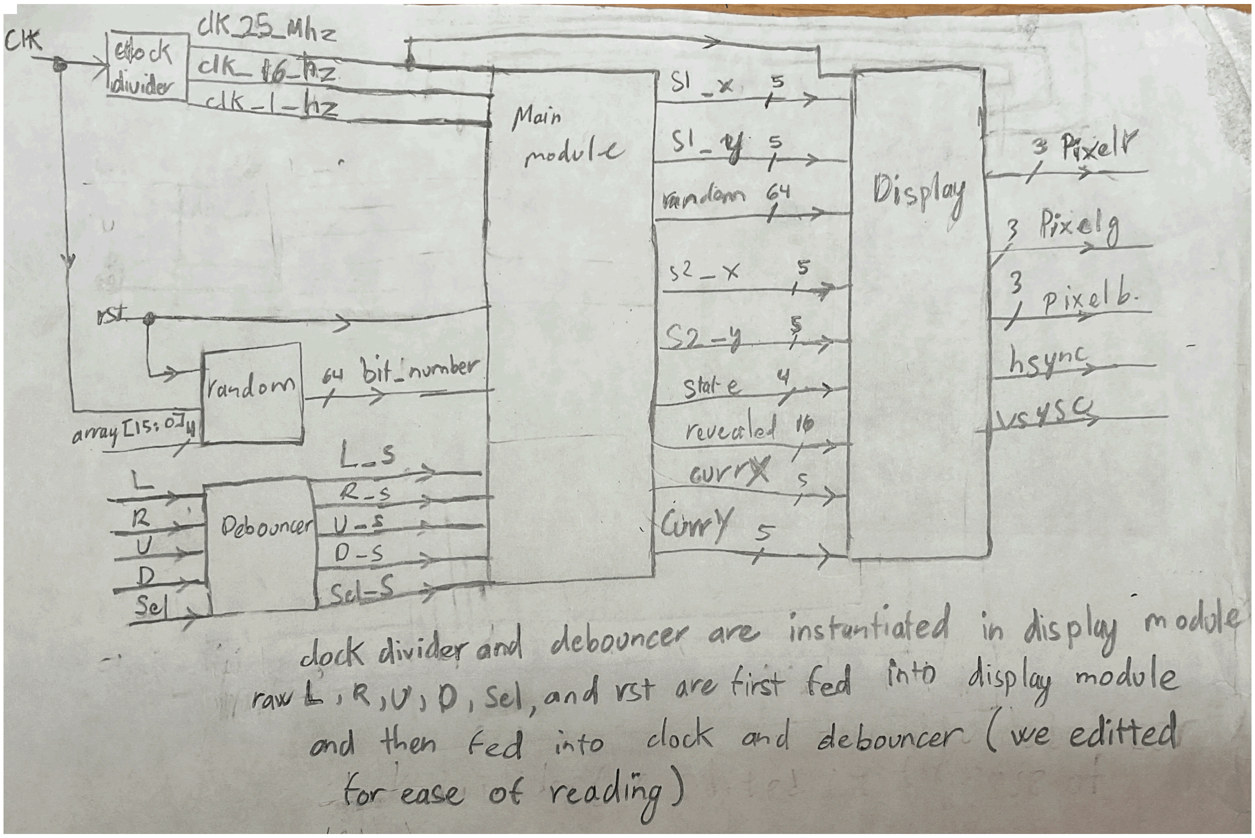
Team number: 5
Henry Liu 906008284, Tan Nguyen 306282087
CS 152A

Lab 4

Intro

We are designing a simple memorization math game. A game board has 16 squares in total. There are 8 pairs of numbers randomly hidden behind those squares. Players have to find these pairs. Every time players click on a square, the number corresponding to that square will be revealed. The player will click on the next square. If the number they reveal matches with the previous one, they will stay shown. Otherwise the squares will flip over and stay unrevealed. Players will win the game when all the squares are shown. Players have to memorize the seen numbers so that they can find matches. Players can navigate on the board by using 4 buttons: up, down, left, and right. Once they decide to reveal a hidden number, they just need to click on the center button.It is impossible to move outside of the bounds of the board, or reveal an already revealed or selected square. Players also can reset the game by flipping the first switch on the board. They can tell their position by seeing the color of squares. Once a pair is revealed, its color will also change. This serves as a notification to let players know that they have found a match. Furthermore, the color of the entire board will change once players win the game. All these numbers and colors display has been done through VGA.

High-level diagram:



clock divider and debouncer are instantiated in display module raw L, R, U, D, Sel, and rst are first fed into display module and then fed into clock and debouncer (we editted for ease of reading)

Module Creations:

Module 1: generating random numbers(Inputs: reset button clicks, clock. Output: 64 bits random number)

The 64 bits random number is equivalent to a 16-element array, and each element has 4 bits. These elements hold 8 pairs of numbers, which have values from 0 to 7. Whenever players click on the reset button, we shuffle these elements. To accomplish this, we orderly go through indices from 15 to 0. At each index, we switch its value with a randomly different index's value.
A random index is generated by a linear-feedback shift register. This shift register is initially loaded with the value of 1. At every occurrence of clock positive edge, we shifted the register's value to the left by one bit. And the entering bit is computed by XOR of some other bits in the register.

Module 2: clock divider(takes 100MHZ clk from display top level module)

I: 100MHZ clock
O: 3 clock frequencies clk_50(later switched to ~16hz),clk_1(1hz),clk_20MHZ(later switched to 25MHz). over the course of development, frequencies were altered to better suit the modules.

We used a counter that counted amt of 100MHZ cycles, for every positive edge, we had counter increment by 1.

We had three clocks:

1. 25MHZ clock, used to update the screen. If the counter was a multiple of 2, it flipped a bit(one +,one - edge). That way every cycle was 4 cycles of 100MHZ clock

2. ~16HZ clock, used to update state and check input. If the counter was a multiple of 3,000,000, it flipped a bit(one +,one - edge). That way every cycle was 6,000,000 cycles of 100MHZ clock

3. 1MHZ clock, this is used to count time elapsed. If the counter was a multiple of 50,000,000, it flipped a bit(one +,one - edge). That way every cycle was 100,000,000 cycles of 100MHZ clock

Module 3: main module(uses IO from random module, from clk module, outputs to display)
This is the main FSM that controls state and abstract concepts of what it means to be a revealed square, track wins, track matches, track movement and what square the user is currently on.
I:clk_16hz,clk_1,clk_25MHz,rst switch, debounced L,R,U,D,sel button clicks
O:S1_x,S1_y(the xy index of which square the user first selected), randoms(the 64 bit representation of which square contains what num, with 4 of 64 bits allocated per square), S2_x,S2_y(x,y index of which square the user selected second),state variable, revealed(a 16 bit rep of which squares are revealed and which are not revealed).

note: revealed[i] == 1 controls whether square at index i is revealed or not
state 1: default state, zero squares are selected
state 2: if one square is selected
state 3: if two squares are selected, and they match
state 4:if two squares are selected, and they dont match

At start or rst:
Set a random variation of (0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7) that corresponds with the 64 bit random input. Then, set the board to start position and set revealed to be all 0's, and set sel_1x and y,sel_2x and y to 0.
note: because of the base 1 index if selx = sely = 0 that means no square is selected.
note:Sometimes the module will use 0-15 indexes or split this up into x and y components.

Every ~16 hertz it will

1. Check if state is 3 or 4, and pause for 10 cycles to display results then set all sel1,2 to 0, set the selected squares to be unrevealed too if state is 4 and it wasn't a match. Return to state = 1 afterwards.
2. Check if a L,R,U,D button was clicked, and update currX and currY to reflect this. Won't allow movement outside of the interval [1,2,3,4], it is 1-base indexed.
3. If user clicks sel on a square, check if it is already revealed,
    a. if state is 1, set S1_x and y to be currX and Y and show to display it is selected.
    b. if state is 2, set S2_x and y to be currX and Y and show to display, furthermore check if the two selected squares match and send to 3 if so, or else send it to state 4

Module 4:VGA Display(top level module)

I: clk_100,rst,L,R, U, D,sel, raw button clicks and clock
O:pixelr,pixelg,pixelb(3 bit rgb code),hsync(move to next pixel horizontally),vsync(move to next pixel vertically)

Takes the FSM's abstract inputs and converts it to raw pixel format to display

Process:

At the start, keep an array called "matches" of size 16 to determine the num associated with each square and set it to the output of main module random output. We split the 64 bit random into 16 4 bit numbers. Every rst, we shuffle the numbers for every square.
Also encode each number from 0-9 into a 2d map of pixels
ex. 0 =

```
49'b
1111111_
1000001_
1000001_
1000001_
1000001_
1000001_
1111111;
```

Every 25MHZ cycle, use currV and currH to keep track of the V and H that is to be set at that time. For every clock cycle, add 1 to current currH, and if it passes the H front porch, set the hsync to 1, to show that a whole line has been scanned and also set currH = 0.
If currV passes V front porch, it shows that all lines are scanned and there is a reset to the start, so we set currV = 0,currH = 0.

This tells the display the order with which to set the pixels and syncs currV,currH to the display.

At the same time, there is a combinational element that tells the circuit what to set for each pixel at each currV,currH.

First, it checks if the current currV,currH is one of the V or H coords on the 4x4 board with this if statement.

```
if(currV>=bYPos &&((currV-bYPos)%total<=
squareL)&&(((currV-bYPos)/total)<5)&&(((currV-bYPos)/total)>0))
          begin
            if(currH>=bXPos &&((currH-bXPos)%total<=
squareL)&&(((currH-bXPos)/total)<5) &&(((currH-bXPos)/total)>0))
             begin
```

Also use (currV-bYPos)/total) (currH-bXPos)/total) with total = the pixel width of a square to find the index of the square. It uses a 1 based index.

Indexes for 4x4 board
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

If the current pixel is inside a square there are six cases:

case 1 all squares are revealed the user has won
case 2 the square is currently the first selected square(i.e x,y index = S1_x,y)
case 3 the square is currently the second selected square (i.e x,y index = S2_x,y)
case 4: the current square is the one the user is currently on (but has not selected or is not revealed)
case 5: the square is revealed
case 6: the square is unrevealed

In which case set the rgb to reflect each of these cases.

If the square is revealed or selected, there is also code to display the number in the middle of the square. First, isolate a 7x7 bit box in the middle of the square. Use matches, which maps the square's index to the num it corresponds to, and bits0_9 which matches a num 0-9 to the actual bitmap of it. For every (i,j) of that 7x7 box, refer to bits0_9 to see if the pixel should be activated(is part of the bitmap).

we check this to see if we should set the number in the middle:

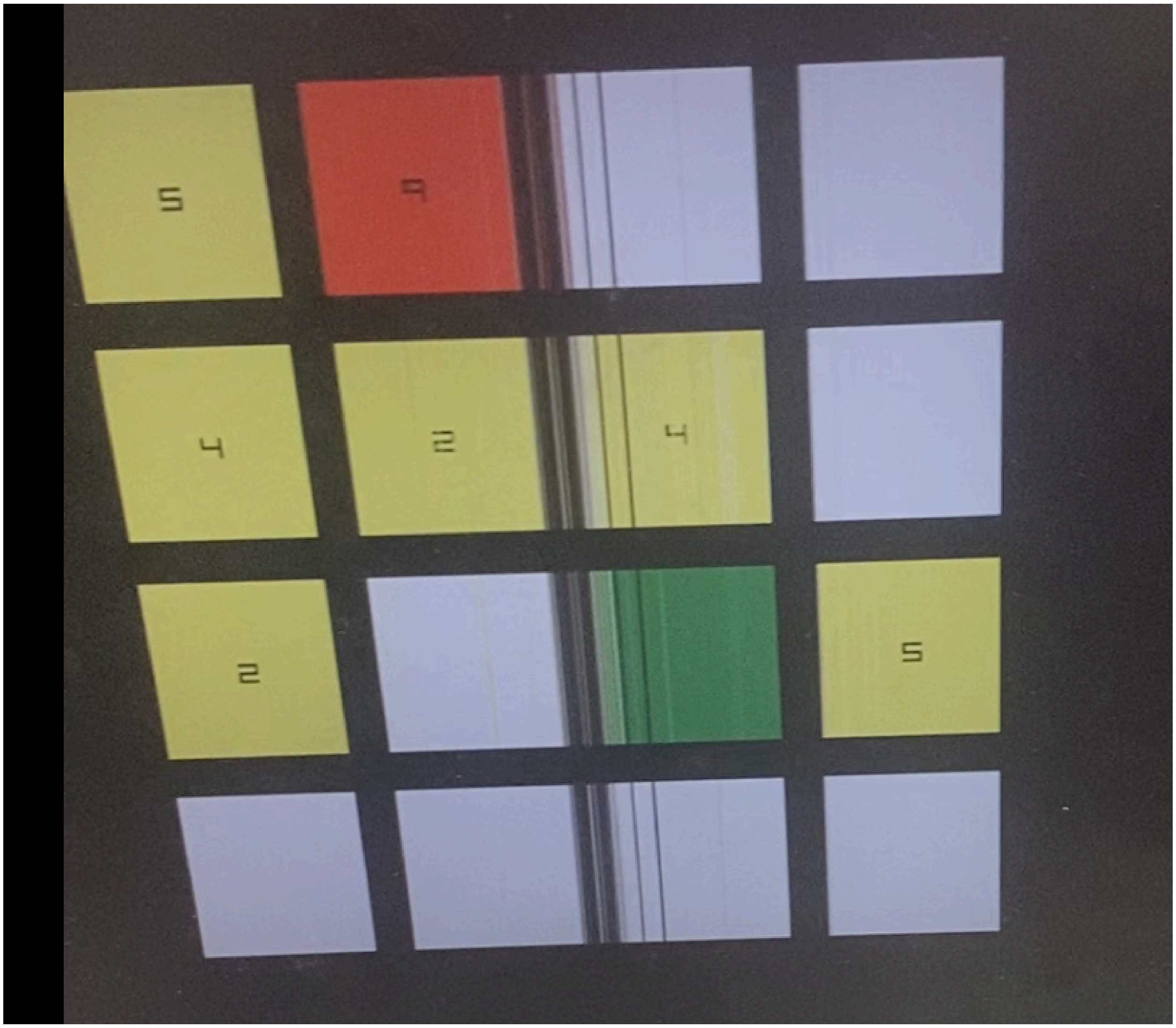
```
(
((currV-bYPos)%total>=30  )  && ((currV-bYPos)%total< 37  ) &&
((currH-bXPos)%total>=30  ) &&  (((currH-bXPos)%total)<37) &&
(bits0_9[matches[(((currV-bYPos) /total)-1)*4+ ( (currH-bXPos) /total )
-1]][((currV-bYPos-30)%total)*7+((currH-bXPos-30)%total)] ==1))
```

Module 5: Debouncer
I:clk, L,R,U,D sel

O:L,R,U,D,sel all debounced

Debouncer takes a ~16 hz clk and raw inputs and saves a 2 bit history for each button. It only sets debounced L,R,U,D if it detects a + edge and in the previous cycle the input equals 0.

A picture of the results:



Tests/Process

We created the project incrementally:

1. We created code display a blank screen on the display
    a. tested the code on sim to see if values were valid
2. We created a static display that shows a 4x4 board of boxes

3. We created a main module FSM that takes L,R,U,D commands, converts it to an index(1-16) for the square the user is currently on, sends it back to the display module so the display knows which square the user is on.
   At this point, there is a 4x4 board the user can move L,R,U,D on. But all squares are blank and have no nums.
   a. tested the bounds to make sure user couldn't move off board
   b. moved L,R,U,D, had a small temporary error where the up button caused the user to move down and vice versa.
4. Implement the FSM states to move from states 1-2-3 for a click of the sel button, and every time a user clicked sel for two squares both the squares became revealed
5. Add numbers to each square, code the display to accept a static order of numbers for each square. At this point, every time the board was loaded it would always display the same order [0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7].
   a. encountered calculation errors as it was hard to isolate a 7x7 box and know what direction to have the bitmaps put in(could display a upside down 4 or display the num at a unexpected place)
6. Add a revealed element to the modules, so that all modules start off unrevealed and when clicked will reveal what number they will contain.
   At this point, the user is able to select two squares, after which the squares will reveal themselves for a set time(10 ~16 hz cycles), and stay revealed for the rest of the round.
7. Create a rst button and have the main module also know when to reset all its state at the start.
8. Implement state = 4 and actually check for matches between squares, if they don't match set them to be unrevealed
9. implement a debouncer
   a. started with a really slow check of 2hz, added a debouncer and realized that 2hz checked for a valid click too slowly, so we increased hertz to ~16 so that although it checked quickly for button clicks, the debouncer checked a 2 bit history and for a + edge, and eventually proved effective
10. Implement a random number:
11. Check for a win with the revealed = 16 1's, and set the display to be green to show a win.
12. At the end we wanted to add a time elapsed module to show the user how much time they took, and indeed added a 1hz clock, but did not have enough time to implement the display of the time elapsed.

Hardest parts of the process:

1. to actually display a blank screen to display, as hsync and vsync had to be set

to reflect the screen.

2. create a combinational circuit that isolated 16 boxes, added some space between them without a hardcode of each box's square
3. match the FSM abstract indexes with actual display positions, as it required many calculations of pixel positions.
4. To actually display the bitmaps, each square had to be further broken down to a small box that displayed the number, and the module had to know how to use the raw currV and currH values to actually iterate the 7x7 pixel box. We had multiple inaccurate patterns(upside down 6, 2 reflected over y axis) because it was indexed incorrectly.

Conclusions

Overall this lab required us to split up a machine into a display module that dealt with the peripheral physical modules, and construct an abstract FSM layer over it. We had to connect each machine and sync clocks so that the display would always feed the correct button clicks and clk to the main module, and the main module would always return state and board info.

We learned that to find errors in projects, it is important to code each feature one at a time. And though we frequently had to recode multiple modules to fit in the new feature, we were at least confident that the last feature worked.We tried to make every module as flexible and scalable as possible. For example, we at first used 0 based index because it was easy to implement, but realized that there would have to be a S1,S2, and there must be a default state that meant(the user has selected no squares). So all indexes were moved to base-1 so that (0,0) could be used to show some sort of unrevealed state. The combinational circuit was made scalable in that we used if statements to isolate each square, and could therefore construct nested if statements to control behavior inside each square. We also added an extra bit to all registers that stored state to account for possible extra data in the future we had to save. It was important to isolate what module did what. For example, it would seem reasonable to detect wins with the main module, but we found it was a lot easier to check from the display whether all squares were revealed and set the display to display a win.