Final Project Report

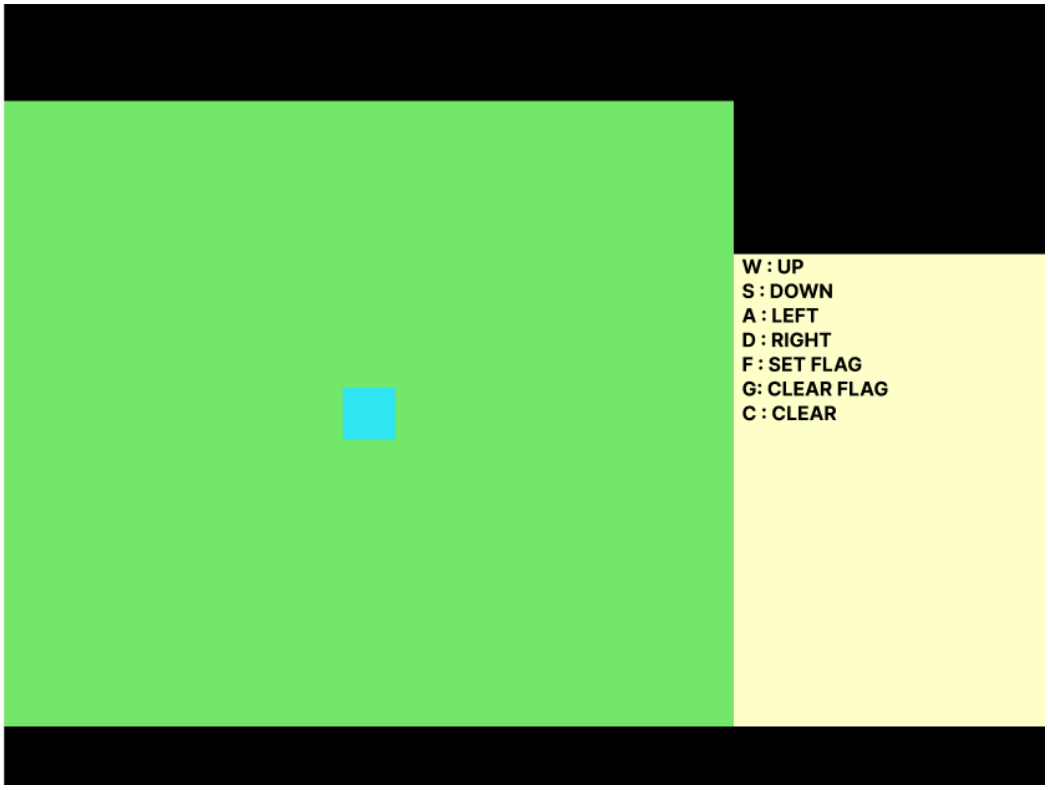
Introduction

This final project implements a basic level of the puzzle game, Minesweeper. This project implements VGA Text Avalon Interface from Lab 7 along with incorporating the MAX3421E file from Lab 6 to connect the USB driver from the NIOS-II to SPI peripheral. The USB driver defines the connection between MAX3421E and the USB keyboard and transmits the data while VGA is used to transform the signal to the monitor and draw each pixel from left to right at a screen refresh rate of 60 Hz. Combining these new elements taught in lab 6 and lab 7, along with implementing game state logic and other technical features, we were able to create a fully functioning basic game of Minesweeper.

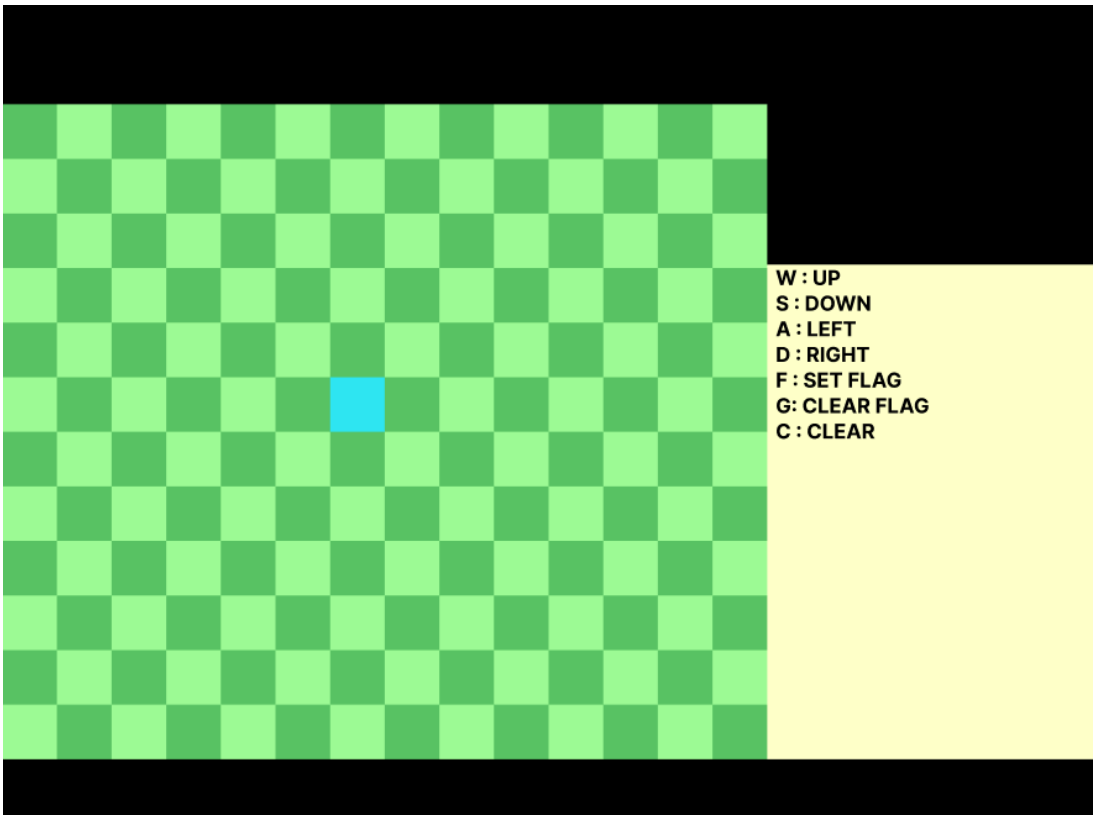
Written Description

The game can be broken down into the following step:

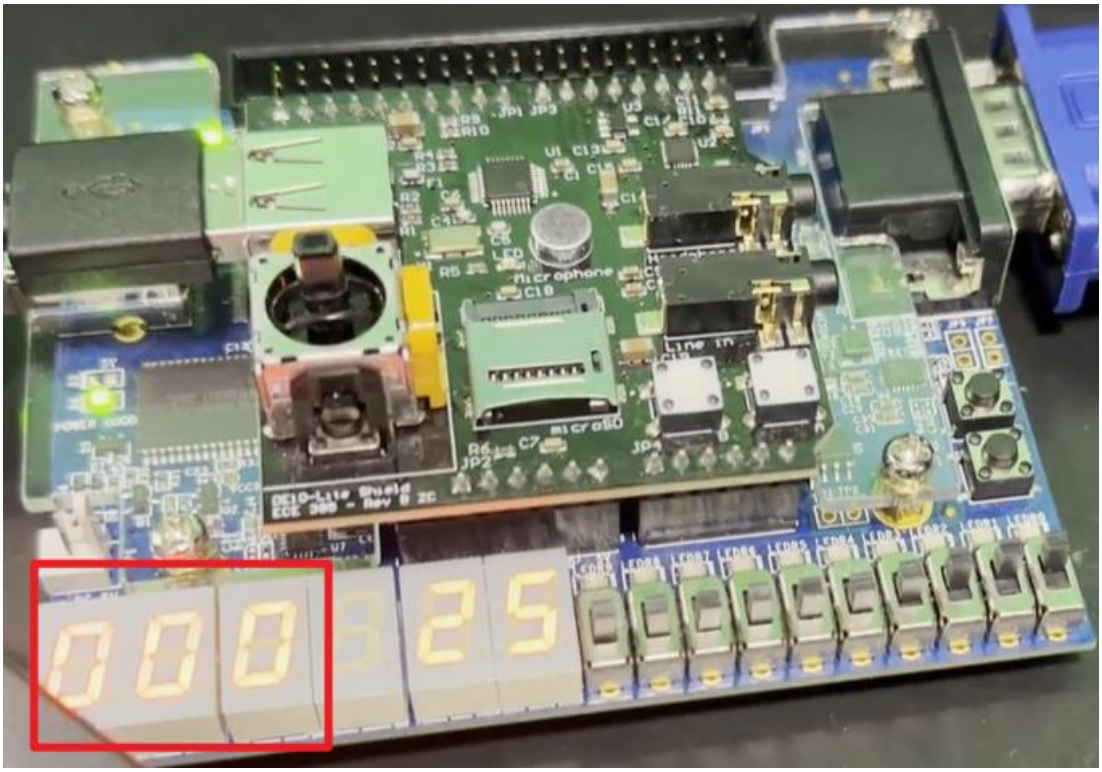
1. The start screen is a green canvas on the left side with a blue rectangle cursor on the center and instructions displayed on the right.



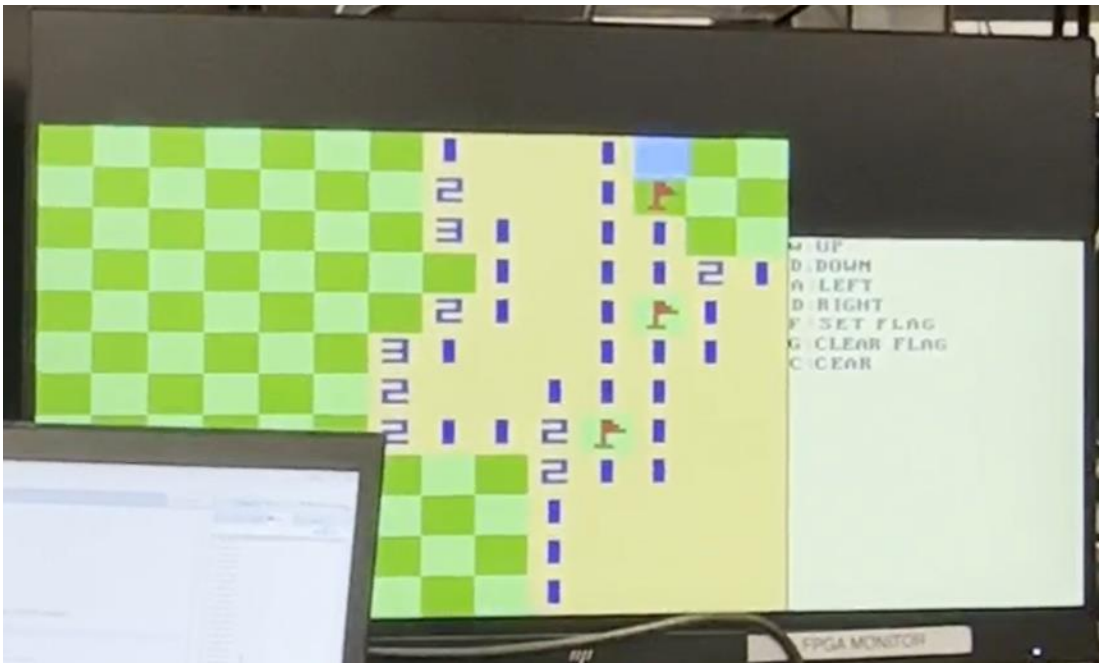
2. The game starts after the user hits enter. The green canvas turns into a green 12X14 checkerboard, and all the keyboards are enabled. Each individual grid is made up of 32X32 pixels, which made the entire check board 384X640 pixel.



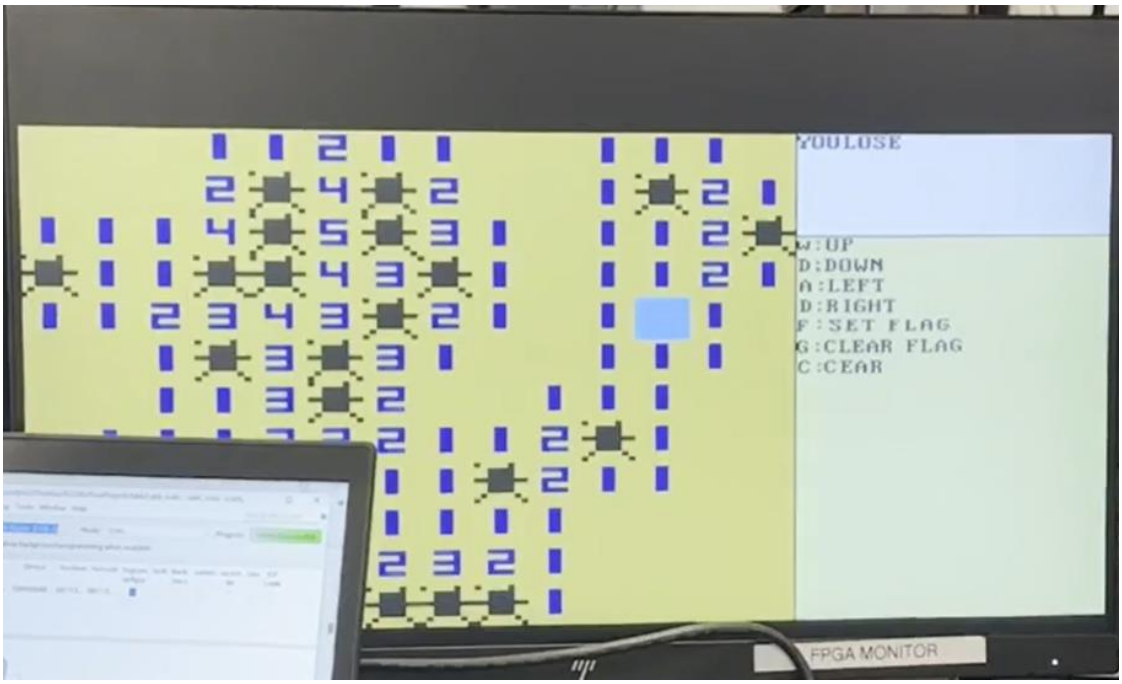
The timer on the FPGA board starts and shows up on hex display along with the number of flags left to place.



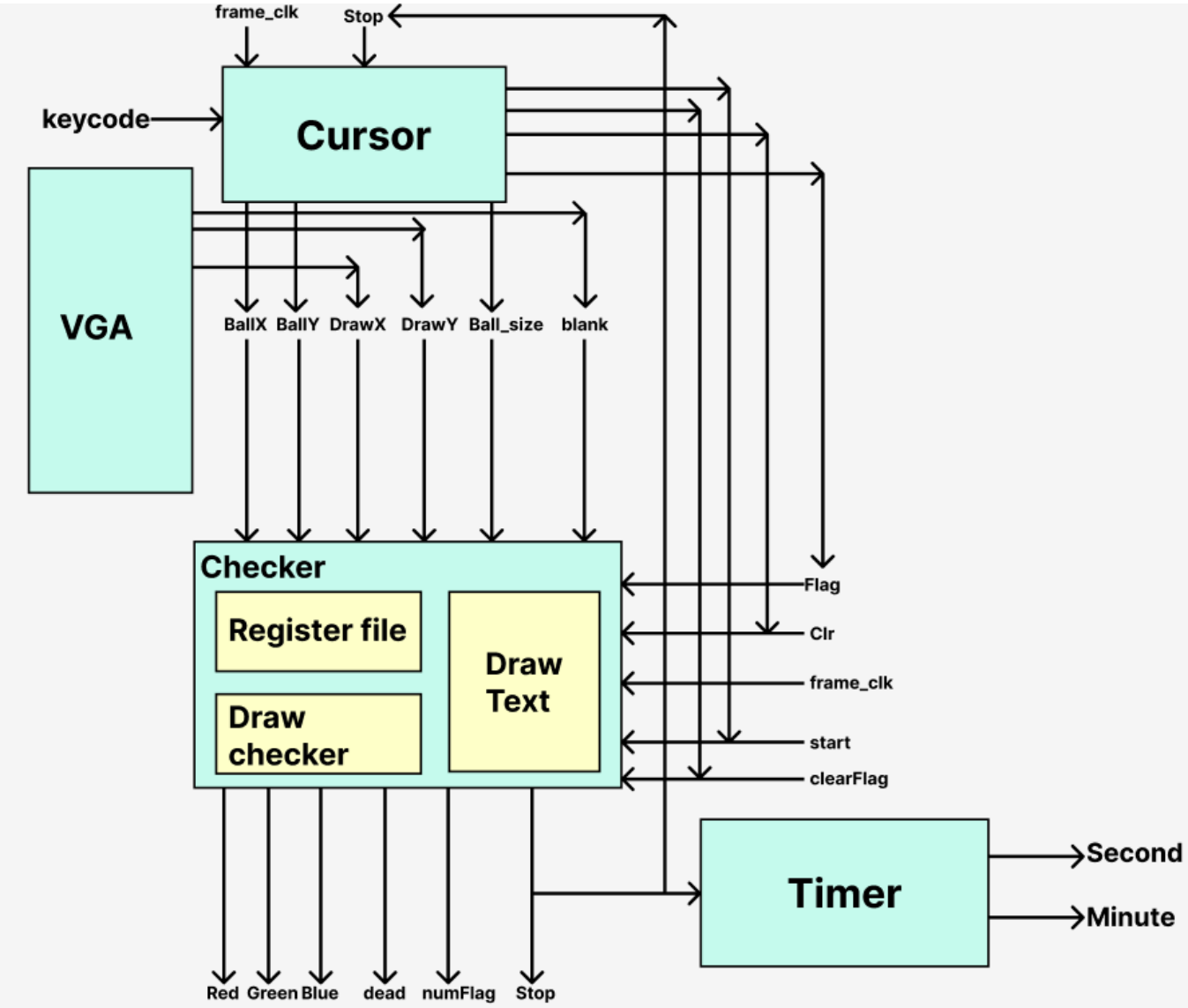
3. User can perform the following operations:
    a. Move the cursor up, down, left, right using AWSD on keyboard.
    b. Place a flag on the position of the cursor by pressing F.
    c. Remove the flag on the position of the cursor by pressing G.
    d. Clearing a space on the position of the cursor by pressing C.



4. Users can win the game by placing all 25 flags onto the correct position where the mine is located. And the user loses the game by clearing a space where the mine is located.

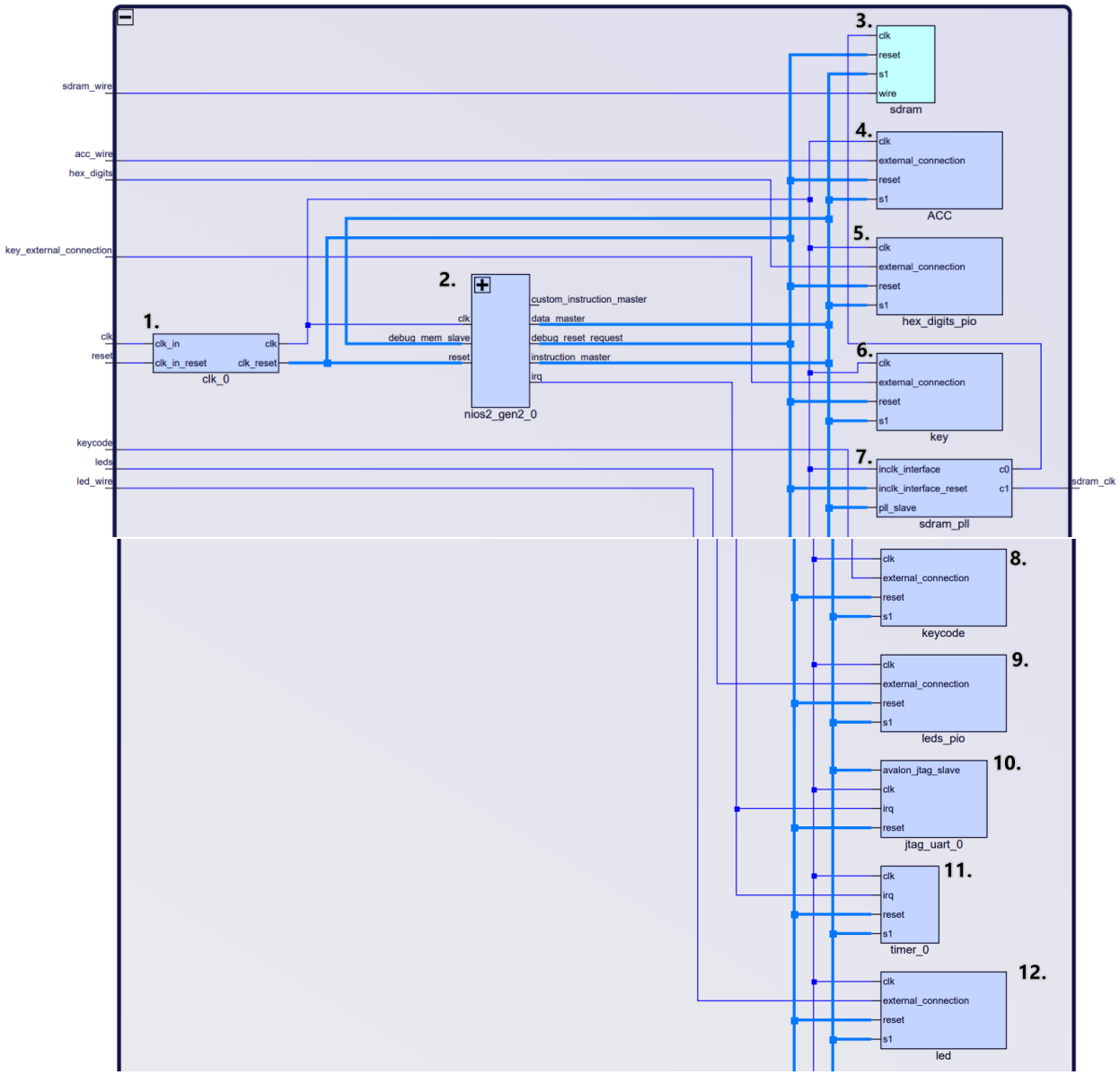


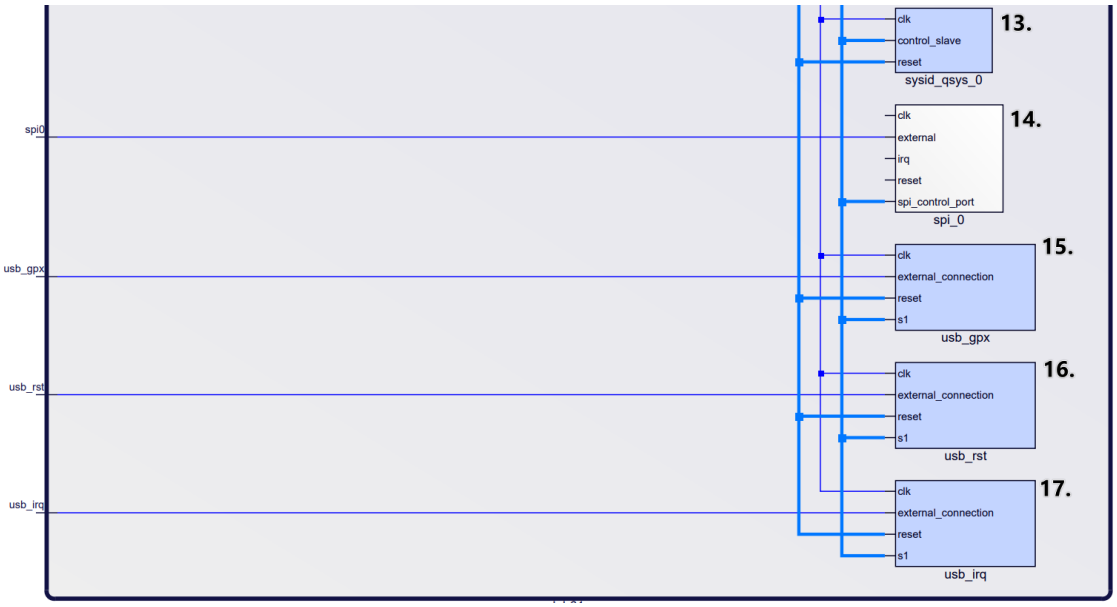
Block Diagram



The checker module contains a register file that contains 168 eight bits register, which is used to store values for each individual grid inside the 12X14 checker.

System Level Block Diagram


sdram_wire
acc_wires
hex_digits
key_external_connection
clk
reset
keycodes
leds
led_wires
sdram_clk
1.
clk_in
clk_in_reset
clk
clk_reset
clk_0
2.
clk
debug_mem_slave
reset
custom_instruction_master
data_master
debug_reset_request
instruction_master
irq
nios2_gen2_0
3.
clk
reset
s1
wire
sdram
4.
clk
external_connection
reset
s1
ACC
5.
clk
external_connection
reset
s1
hex_digits_pio
6.
clk
external_connection
reset
s1
key
7.
inclk_interface
inclk_interface_reset
pll_slave
c0
c1
sdram_pll
8.
clk
external_connection
reset
s1
keycode
9.
clk
external_connection
reset
s1
leds_pio
10.
avalon_jtag_slave
clk
irq
reset
jtag_uart_0
11.
clk
irq
reset
s1
timer_0
12.
clk
external_connection
reset
s1
led



Module Description

1. Cursor.sv

Inputs: Reset, frame_clk, stop, waitsig,

[7:0] keycode

Outputs: [9:0] index_X, index_Y, CursorS,

setFlag, clearFlag, clear, is_start

Description: This module passes in a reset and frame clock signal, along with keycode I/O, and outputs the coordinates and the size of the cursor. The position of the cursor is updated on the positive edge of the clock in an always_ff statement, in which we have specific keycodes to move the cursor position by one square.

2. Checker.sv

Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball_size,

blank, flag, clr, frame_clk, start, clearFlaggg,

Outputs: [7:0] Red, Green, Blue, deadd, numFlagg,

stop

Description: The checker module takes in the position of the cursor and the pixel that is supposed to draw out and outputs the rgb color corresponds to that pixel. The pixels covered by the square cursor should be blue and the rest of them should display the game image. Flag, start, and clearFlaggg signal is received from the cursor module and used in content.sv that exist inside the checker module for manipulating the data based on user input.

3. vga_controller.sv

   Inputs: Clk, Reset

   Outputs: hs, vs, pixel_clk, blank, sync,

   [9:0] DrawX, DrawY

   Description: The vga controller module takes in clock and reset signal and outputs horizontal and vertical sync pulse, 25 MHz pixel clock output, blanking interval indicator, composite sync signal, and 10-bits xy coordinate.

   Purpose: This module is used for producing timing signals and keeping track of the coordinate position of each pixel on the VGA monitor.

4. Timer.sv

   Inputs: clk, stop

   Outputs: [7:0] sec, [3:0] min

   Description: This is used for timing purposes. Once the game starts, it will start timing and outputs the time in second and minute. These two values will be displayed on he hex display of FPGA. The timer will stop once the stop signal is high, which is when the user win or lose the game.

5. Content.sv

   Inputs: [31:0] indexxx,

   [9:0] cursorx, cursory,

   flag, frame_clk, start, clearFlag,

   Outputs: [7:0] value_out, deadd, numFlag, win

   Description: The way data are stored in this game is by creating one 8 bits register for each individual grid on the 12x14checkerboard. This module is used for storing all the data of 128 registers. It takes in the index of the position of a certain register and outputs its corresponding value. It also takes in flag, start, clearFlag signal as well as the position of the cursor to achieve the functionality for the game by modifying the data that is stored inside these registers. How values are modified is specified in reg_sv.sv.

6. reg_8.sv

   Inputs: Clk, Load, clear, start, clearFlag, END,

   [7:0] D,

   Outputs: [7:0] Data_Out, dead

Description: This is a module of one eight bits register. In addition to storing the eight bits value, this module also modify the value based on Load, clear, start, clearFlag, and END signal:

If the END signal is high, that means the game is over and the register will output its original value so that corresponding image could displayed on the monitor.

If the start signal is low, that means the game has not started yet and the register will output value that represents green color.

If the flag signal is high, that means a flag will be placed onto the corresponding grid and the register will output the value that represents the flag.

If clearFlag signal is high then the flag will be removed, and the register will output value that represents green color again.

If the user tries to turn clear signal high to a register that stores value that represents a mine, the user loses the game, and the register will output a positive dead signal back to check.sv.

7. DrawGrid.sv

   Inputs: [7:0] Grid,

   [4:0] gridx, gridy,

   [2:0] index_x, index_y,

   Outputs: [7:0] Red, Green, Blue

   Description: DrawGrid.sv takes in the 8 bits value of a register, index position of that register, and the position of the pixel that supposed to draw and returns it's corresponding rgb value. The conversion are as follows:

   1) If the 7^{th} bit of the value is 1, the module will draw a flag on corresponding grid that looks like this: 

   2) If the 6^{th} bit of the value is 1 and the second bit is 0, the module will draw a mine on corresponding grid that looks like this:  .

   3) If 3-5 bits have some value other than 000 and the second bit is 0, the module will draw the number that is represented by these bits.

4) If the second bit is 1, the module will draw the color of the checkerboard, which is just green.

This module only determines the color choice of the corresponding grid. The actual drawing part is done in font_rom.sv, which exists inside DrawGrid.sv

8. font_rom.sv

   Inputs: [2:0] indexx, indexy,

   [2:0] number,

   flag, mine, dead,

   Outputs: pixel

   Description: The font rom module is implemented the same way as lab 7, except this time the module takes in some parts of the value of the register (specifically 2 to 7 bits), breaks them done into specific signal (number, flag, mine, dead) and outputs the drawing of the corresponding grid based on the input index. Each 32x32 pixel grid is further broken down into 64 four bits grid so that each 4x4 grid only displays one color.

   For example, an image of flag is represented in this way:

```
8'b00011000;
8'b00111000;
8'b01111000;
8'b00001000;
8'b00001000;
8'b00011100;
8'b00000000;
```

   Here's how numbre three is represented:

```
8'b00000000;
8'b00111100;
8'b00100000;
8'b00111100;
8'b00100000;
8'b00111100;
8'b00000000;
8'b00000000;
```
(Those are mirror image since VGA will draw everything in reverse)

9. DrawText.sv

   Inputs: [9:0] DrawX, DrawY,

   win,dead

   Outputs: on

   Description: Same as DrawGrid.sv, this module is used for displaying the instructions of the game and the game result on the right side of the screen. DrawX and DrawY are the

index position of the corresponding index. Win and dead signal are for telling the module if you win the game or lose the game. On signal is the output for determine if the pixel is using foreground color or background color and this is received from font_romtext.sv.

10. font_romtext.sv

    Inputs: [2:0] indexx, indexy,

    win, lose,

    Outputs: on

    Description: Same as font_rom.sv, this module is used for drawing out the text instruction for the game. The output signal ‘on’ decides if the pixel uses foreground color or background color based on its corresponding position and if the user wins the game or loses the game.

11. Lab62.sv

    Inputs: MAX10_CLK1_50,

    [1: 0] KEY,

    [9: 0] SW

    Outputs: [9: 0] LEDR,

    [7: 0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5

    DRAM_CKE, DRAM_ADDR,

    [1: 0] DRAM_BA,

    [15: 0] DRAM_DQ,

    DRAM_LDQM, DRAM_UDQM, DRAM_CS_N,
    DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N,

    Description: This is the top-level module for this game, which includes the module for cursor, VGA, Checker, and Timer.

Documentation of Design Resources and Statistics

LUTs	7385
DSP	0
Memory (BRAM)	11264 bits
Flip-Flop	3055
Frequency	69.83 MHz
Static Power	96.63 mW
Dynamic Power	77.80 mW
Total Power	204.16 mW

Conclusion

Our design was successful, and we had the key aspects of what we wanted to incorporate into the game successfully implemented. We borrowed some components from previous labs, specifically lab 6 and lab 7, in which we implemented a VGA Text Avalon Interface and included a USB driver to connect the keyboard input through NIOS-II. We were able to follow our initial plan and roughly followed through our timeline, in which we took our project one step at a time, i.e., we started out with something simple as drawing a 12x14 grid, implementing a cursor, drawing out the flags, mines, and numbers, and then we added the finishing touches afterwards, including some semi-advanced technical features such as a stopwatch. Debugging our project was relatively simple as we did not need to use ModelSim or SignalTap, as most issues stemmed from our game state logic, which were usually resolved from tracing through our code. We implemented all the basic features required, and added a couple more advanced features as stated before, however, there was one advanced feature we were not able to successfully implement due to lack of time and a lot of complications, which could have given us a couple more difficulty points if we could have incorporated it into our final project. This included random map generation, which would have increased the difficulty of our project by a significant margin, as we would need to utilize of Linear Feedback Shift Register to generate pseudo-random numbers in which the mines would be randomized accordingly. Doing so would also require creating and instantiating far more modules including an entirely new logical aspect in which the code itself would have to figure out the game logic based on the randomized mine placements to display for players, along with some search algorithm for when the player clears a large chunk of the board. This was the only major advanced technical aspect we were not able to overcome.