GOMOKU

Walk through and Final Report

Lab 3165, Thursday Evenings @ Station 63/64

University of Toronto

December 5, 2019

WHY GOMOKU?

Gomoku is a board game where each step is built upon previous choices which means to implement this game, we will need to store states. We will need to build a sequential circuit with registers. At the same time, the circuit will need to be clocked, which is again, related to CSC258 material. The game state(game board) will be displayed on a screen through a VGA cable and this is related to the Lab7 material. We will also use the hex display to show some of the stats for the ongoing game.

Project Milestones

- Display the empty board on the screen through VGA cable. Go over the resources posted on quercus regarding keyboard control and then implement the keyboard control. Find resources for the while/black token and the game board.

- Users should be able to move the pointer around on the screen using the arrow keys on the keyboard. Users should also be able to put the token down (using the enter key on the keyboard) on any valid position on the game board. Notice that tokens have alternating colour, meaning that if the last token put down was a white one the next one should be black.

- Check if the game has been won or not. If won, by which player. Display a message for who won the game. Make the hex display various game stats, including the current player identified by the player's number.

Worth Mentioning Features

- Support of keyboard control through the PS2 interface.
- Moving cursor on the screen to indicate the current location of the cursor. No more looking at hex and counting on the grid!
- Automatic alternating colour switching, avoids cheating in game.
- Placing a stone at an already occupied location is forbidden.
- Automatic judging of winner, more fair!

Implementation Details

Structure of the project

- `GOMOKU_FPGA/adapters/*`
  - Contains PS2 keyboard adapters, courtesy of Alex Hurka, link provided by professor on Quercus, under Project Proposal + Resources section.
  - Contains VGA Adapters, borrowed from Lab 7 of CSC258.
- `GOMOKU_FPGA/DE1_SoC.qsf`
  - Required pin assignment file for the FPGA board, provided by professor on Quercus.
- `GOMOKU_FPGA/gomoku.v`
  - Actual Implementation of the Gomoku Game, top level instantiation name is `gomoku`. (This is code written by us.)
- `GOMOKU_FPGA/utils/*`
  - Contains utility files, including the `bmp2mif` written in C provided.

Implementation Details

Input using the PS2 Keyboard

- `<space>` for starting the game
- `a` for moving toward left by one position
- `s` for moving down by one position
- `d` for moving right by one position
- `w` for moving up by one position
- `enter` for placing the stone at the position where the cursor is currently at.

Implementation Details

Output Data on HEX and VGA display

- `HEX0` for y location on the grid, from 0 up to 6
- `HEX1` for x location on the grid, from 0 up to 6
- `HEX4 & HEX5` for the number of steps taken to win the game. Users can compare the number of steps taken to win the game!

What did we do? (Main Components)

FSM that draws the screen

```
106     always@(posedge CLOCK_50)
107     begin
108         if (has_begin) begin
109             writeEn <= 0;
110             if (~resetn) begin
111                 vga_x <= 8'b0;
112                 vga_y <= 7'b0;
113             end
114             else begin
115                 writeEn <= 1;
116                 if (vga_x != 159 || vga_y != 119) begin
117                     if (vga_x == 159) begin
118                         vga_x <= 8'b0;
119                         vga_y <= vga_y + 1;
120                     end
121                     else begin
122                         vga_x <= vga_x + 1;
123                     end
124                 end
125                 else begin
126                     vga_x <= 8'b0;
127                     vga_y <= 7'b0;
128                 end
129             end
130         end
131 
132     end
```

What did we do? (Main Components)

Choosing the colour

```
always@(posedge CLOCK_50)
begin
    //  ! RECALL DEFINITIONS
    //  reg [119:0] black_canvas [159:0];
    //  reg [119:0] white_canvas [159:0];
    //  these numbers are set to prevent overflow
    for (xc = 5; xc <= 154; xc = xc + 1) begin
        for (yc = 5; yc <= 114; yc = yc + 1) begin
            if (black_mask[xc-1][yc-1] == 1 || black_mask[xc-1][y
                black_canvas[xc][yc] <= 1;
            else if (white_mask[xc-1][yc-1] == 1 || white_mask[xc
                white_canvas[xc][yc] <= 1;
        end
    end
    if (empty_board[vga_x][vga_y] == 1'b0)
        vga_colour <= 3'b110;
    else
        vga_colour <= 3'b000;
    if (white_canvas[vga_x][vga_y] == 1'b1)
        vga_colour <= 3'b111;
    else if (black_canvas[vga_x][vga_y] == 1'b1)
        vga_colour <= 3'b000;
    if (winner_txt[vga_x][vga_y] == 1'b1 && has_win) begin
        if (white_win)
            vga_colour <= 3'b111;
        if (black_win)
            vga_colour <= 3'b000;
    end
    // red pointer this should be done last
    if (vga_x == 33 + 16 * in_x && vga_y == 13 + 16 * in_y)
        vga_colour <= 3'b100;
end
```

What did we do? (Main Components)

Keyboard Controller

```
// keyboard control
reg left_lock, right_lock, up_lock, down_lock; // simulate pulses
wire left_key, right_key, up_key, down_key;
assign left_key  = a && ~left_lock;
assign right_key = d && ~right_lock;
assign up_key    = w && ~up_lock;
assign down_key  = s && ~down_lock;
always@(posedge CLOCK_50)
begin
    if (~resetn)
    begin
        left_lock <= 1'b0;
        right_lock <= 1'b0;
        up_lock <= 1'b0;
        down_lock <= 1'b0;
        in_x <= 3'd3;
        in_y <= 3'd3;
    end
    else begin
        left_lock <= a;
        right_lock <= d;
        up_lock <= w;
        down_lock <= s;

        if (left_key)
            in_x <= in_x == 3'd0 ? 3'd0 : in_x - 3'd1;
        if (right_key)
            in_x <= in_x == 3'd6 ? 3'd6 : in_x + 3'd1;
        if (up_key)
            in_y <= in_y == 3'd0 ? 3'd0 : in_y - 3'd1;
        if (down_key)
            in_y <= in_y == 3'd6 ? 3'd6 : in_y + 3'd1;
    end
end
```

What did we do? (Main Components)

Placing Stones

```
// go is the signal for placing the token
// embedded logic for checking if the position has been occupied or not.
assign go = (enter && ~white[in_x][in_y] && ~black[in_x][in_y] && ~has_win);

always@(negedge go)
begin
   if (in_color == 1'b1) begin
      white[in_x][in_y] <= 1'b1;
      white_mask[in_x * 16 + 33][in_y * 16 + 13] <= 1'b1;
   end
   else begin
      step <= step + 8'b1;
      black[in_x][in_y] <= 1'b1;
      black_mask[in_x * 16 + 33][in_y * 16 + 13] <= 1'b1;
   end
   in_color <= !in_color;
end
```

Discussion

Did it work?

- Yes! It works well!
- Up to our manual testing by playing against each other there is no bug.
- We tested main components of the project using ModelSim and they behave exactly as what we want : p

Discussion

What did you learn specifically

- Keyboard Control! This was brand new to us and took a long time to figure out how to use the module. It is really cool to be able to control the game using a keyboard.
- In lab7 we encountered the VGA module, however at that time we just followed the instructions and didn't really dive into how things were realized. After the project, we completely understood that VGA has to be drawn with two components, one moving the pointer to different positions and then one changing the colour as appropriate while we loop over the area that we want to draw.

Discussion

What did you learn specifically

- Initial Blocks! This didn't appear in the course material but we find it was a great help in initializing the registers which we used to hold initial values needed for the program.
- For loops! This is not synthesis-able code, but it turns out to be very useful for handling code that are repetitive!

Discussion

What would you do differently next time?

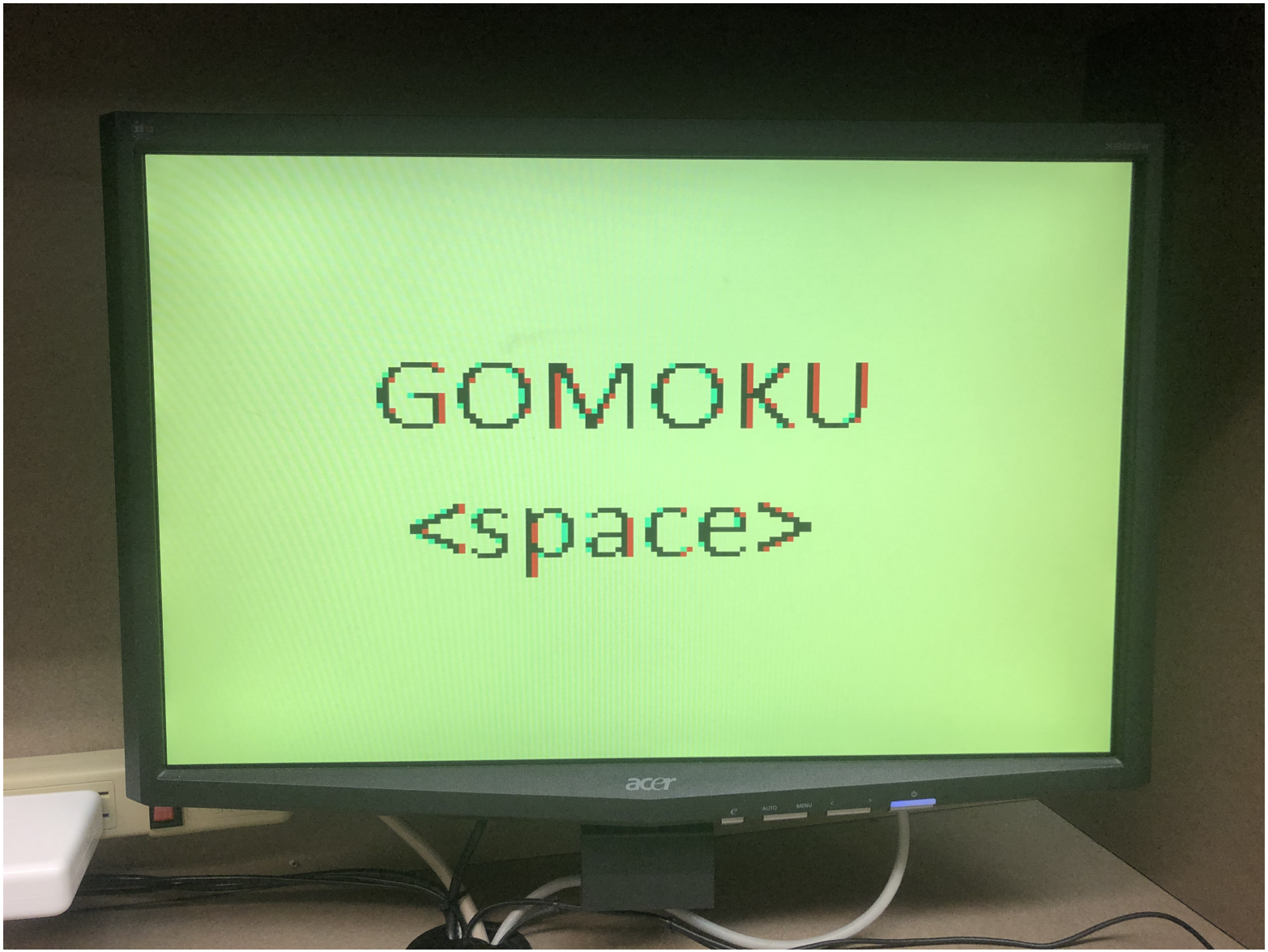
- For a static game like this, maybe it is a better idea to have the screen not redrawn over and over : (
- To be more (computationally) efficient, we can try to figure out what part of the pixels change during each transition and just redraw/overwrite the specific part.
- Use a memory map rather than using arrays to store information about the screen.

Conclusion

- We think the project definitely helped us better understand the course materials and especially verilog!
- It is nice to see what we learnt in class in action using a cool project.
- Until now, ModelSim is still not very intuitive to us and in that sense hardware has been hard for us. However, hardware has been great fun!
- We are thankful to our TA who have been working diligently through out the term in making sure we understand key concepts through out the term. Thank you so much, we really appreciate it!

Time for a little demo!

Start Screen



Game Play and Win!!