ChessAi

Abstract

The goal of this project is to create chess board that acts as an opponent who can be played against with little to no human computer interaction. Physically it consists of a chess board with 64 LED indicator lights, a set of standard chess pieces modified with magnets on the bottom and wiring underneath the board to read off piece positions. On the FPGA we implemented an FSM to check the validity of all moves and to relay moves made by the human to a computer running a python script that determines the opponent’s move. It also included a connection between the computer and the LEDs to provide useful signaling and indications of the opponent's move.

INTRODUCTION

Many people love to play chess but nowadays it’s hard to find people to play with – especially for the senior and more experienced population. A subgroup of those people either don’t own or don’t know how to use a computer, so they cannot play chess online. We decided to build a physical chess board that brings the joy of moving the figures with your hands and the pleasure and flexibility of playing against a computer. The whole system is composed of a figure state detection module (module 1), an FSM that checks the validity of movements and tracks the progress of the game, communication modules with an external computer and an LED system that displays movements the computer wants to do.
As we always need to know the position of every figure, we will build a module to help us do so. We will have 8 horizontal (blue) and 8 vertical (red) wires running under the board with hall sensors connected to the grid formed by the wires. A hall sensor is a digital device that signals the presence/absence of strong magnetic fields in its proximity. Whenever a figure (with a magnet placed on the bottom) is placed on a square, it will produce a signal that is channeled to the FPGA. The horizontal and vertical lines will be connected to 2 multiplexers that will select which lines to analyze. The setup is shown in the figure below. MUX2 will individually supply the lines a-h with a high signal. At the same time, MUX1 will run through lines 1-8 and output a high signal if there is a figure on the corresponding square and a low signal otherwise. Since each mux is connected to 8 wires, we need only 3 bits to control each of them. Since only one hall sensor is powered at a time, only the signal from that particular sensor is relevant to the FPGA. We have determined that disconnected hall sensors (either from the positive or ground source) have a high impedance at their output so that they don’t interfere with each other.

A counter implemented in software runs through 64 permutations activating and deactivating vertical and horizontal wires. The corresponding high and low signal was used by another module that generated a 64-bit long number that shows which squares of the chess board have figures on them. The challenging part here was to synchronize the arrival of the debounced signal received from the hall sensor with the update of the figure state. I solved the problem by always updating the previous square with the readily available signal. That solution worked fine for all of the squares except the starting one (at 0,0 position) since it did not have a predecessor. To solve that problem, I had to make sure that the LED wire system selection algorithm waits for the duration of the debounce for the initial state.

By testing various time constants, we have determined that an individual hall sensor should be powered for at least 0.4ms (SQUARE_WAIT) to produce reliable signals. Thus, the latency of this module is 64xSQUARE_WAIT ≈ 25ms (time spent
looking at every square) This module can have a large latency (<10ms) since faster state updates are not crucial to the whole system, taking into consideration the speed of possible figure movements.

Shown below is a detailed block diagram of the connection between the output ports from the FPGA to the wire system powering the hall sensors. For further implementations I would recommend designing a PCB instead of manually manufacturing the wire system. It required around 700 soldering points and around 5m of wire to get the system running. Mistakes in the manufacturing process were inevitable so that we had to struggle with faulty hall sensors and weak connections, which slowed down our progress and made it impossible to achieve the intended goal of having a fully functional board.

This module can only detect the presence/absence of figures on each square. In order to track different figures, we need an FSM to keep a record of the state of the game.

![Block Diagram](Image)

**Chess Game FSM**

*Includes:* Keeps track of figures, detects changes, tests legality of moves, communicates with other modules and looks out for the winner

While the essentials of chess basically only requires an FSM which keeps track of when a piece disappears and where it reappears, there are many subtleties in both the rules and how the moves are enacted which will need to be considered since we want as little of the computation to be done on the off board computer as possible.
First the FSM needs to ensure that the move order is preserved. This means that if a black piece is picked up after another black piece and before a timer runs out (in the case of capturing or castling) a warning signal is flashed on the LED’s.

Next, once the piece has been placed, the FSM must ensure that the move was a valid one. Naively, this should be an easy task, but again, chess is a complex game with complex rules. Pawns can only move forward once...unless they’ve reached the other side...or they can move twice if they’re on the second row...well except if an opponent could capture them in the intermediate position (en peasant). There are a lot of these oddities which will need to be accounted for in the FSM.

Once the movement has been verified by the FPGA, it will update the state of all the pieces accordingly then send the most recent move to the computer (over UART?) and wait for a response.

**External Computer Attachment - Response of the opponent**

**Includes: communication with the computer + python script**

*Example: send:c2c4:  returns: d2d4*

Due to the complexity of chess gameplay and how much research and effort has been put into making decent chess AI on standard PC architectures, we have decided to leave the opponent move generation to a separate computer running some sort of standard chess library in python. This python script will simply receive moves generated by our system’s FSM and respond to them with optimal opponent moves over UART and an improvised communication channel over a teensy. It is straightforward to implement the communication over UART from the FPGA to the computer: we need to send 12bits (6bits for the initial position, 6bits for the final position) over the already available UART protocol. As the maximum number of bits UART can transmit at one time is 8 bits, we had
to divide the 12bit long signal into two 6bit signals, appended with bits to indicate the order in which they were sent.

The communication from the computer to the FPGA is more complicated. I have used serial communication between the computer and a teensy to control its digital pins. Those digital pins (7 of them) were connected to the jc digital inputs of the FPGA. The first bit indicated weather we are sending the initial or final position of the intended movement and the remaining six bits were used to transmit the coordinates of the positions the chess figures should move.

**MODULE 4 - Opponent’s movement signaling via LED’s**

Since this game is meant to be played against a computer - but on hardware, the simplest way for the computer to control the chess figures is to signal where the figures should be moved (by the player) via means of lighting up LEDs embedded in the chess board. Similar to the Module 1, we have an additional set of 8 horizontal and 8 vertical wires running under the board. They are able to individually power each of the 64 LEDs. Two LEDs turned on and off periodically signal the intended movement: the one from where the figure is to be moved and the second one to where it should be moved. Again, a pair of 8-bit mux-es will be used to turn on/off individual LEDs.

One input to this module is the **show_pos** signal to start displaying the needed movement of the figures. The **from_to** signal is a 12-bit long signal. First 6 bits are designated to signal the coordinate of the “from” LED while the last 6 bits are to signal the “to” LED. The two LED’s will turn on and off alternatively until the player moves the figure.

There are several special patterns displayed by the LED array when:
- An illegal move has been made LEDs will create an X sign
- The game has ended, the winner side of the board lights up
'timescale 1ns / 1ps

module detectAndSignal(input     clk_100mhz,
    input [15:0]     sw,
    input [3:0]      jb,
    input [6:0]      jc,
    input [63:0]     image,
    input           btnc,
    input           btnd,

    //output[63:0] positions,
    output logic [6:0] ja,
    output[15:0]    led,
    output         led16_b,led16_r,led17_g,led17_r,led17_b
);

parameter TIMECONST = 10_000;
parameter SCAN_DELAY = 256*TIMECONST;
parameter SCAN_DELAY_MULT = 8;
logic figure;
logic reset=0;
logic sig_ready;
logic board_state_ready=0;
debounce db1(.reset_in(reset),.clock_in(clk_100mhz),.noisy_in(~jb[0]),.clean_out(figure),.ready(sig_ready));

logic wrong_move;
//assign wrong_move = sw[15];

logic [11:0] from_to = 12'b101_111_100_110;
logic [11:0] computer_out = 12'b101_111_100_110;

logic win_a;
assign win_a = sw[14];

logic win_b;
assign win_b = sw[13];
logic[63:0] positions = 64'b0;
logic[63:0] final_positions = 64'b0;

logic[31:0] counter = 0; //hold counter
logic[31:0] count = 0;
logic[4:0] counterx=0;
logic[4:0] countery=0;

logic scan_enable=0;
logic zero_pos;
logic[31:0] led_timer_big;

always_ff@(posedge clk_100mhz) begin
    //this part scans the board and detects the figures
    if (counter<SCAN_DELAY*{sw[9:0],1'b1}*(1+sw[11:10]*SCAN_DELAY_MULT)) begin

if (~scan_enable) counter <= counter+1;
   end
else begin
   scan_enable<=1;
   counter <=0;
end

if (scan_enable&(count<TIMECONST*{sw[9:0],1'b1}))   //
begin
   count <= count+1;                //
end
else                              //
begin   count <= 0;
if (counterx==7) begin           //
   counterx <=0;       //
   countery <= countery+1; //
   end
else
   counterx <= counterx+1;
if (countery==8)
begin
   final_positions<=positions;
   board_state_ready<=1;
   countery <= 0;
begin
   scan_enable <= 0;       //
else begin
ja[2:0] <= {counterx}[2:0];  //
ja[5:3] <= {countery}[2:0];  //
end
end
//this part builds the scan output
if (~scan_enable &(counter<(SCAN_DELAY)*{sw[9:0],1'b1})
   *(1+sw[11:10]*SCAN_DELAY_MULT))
   &(counter<(99*SCAN_DELAY/100)*{sw[9:0],1'b1})
   *(1+sw[11:10]*SCAN_DELAY_MULT))
begin
ja[2:0] <= {3'b000}[2:0];
ja[5:3] <= {3'b000}[2:0];
zero_pos <=1;
end
else if(zero_pos&~scan_enable)
begin
zero_pos <=0;
positions[0]=figure;
end

if (sig_ready&scan_enable&(counterx+countery*8>1))
begin
positions[counterx+countery*8-1]= countery < 2 ? 1 : figure;
end

//-----------------------------------------------------------------------------
//this part displays the input
//it is blocked by the scan_enable signal
//-----------------------------------------------------------------------------
led_timer_big <= (SCAN_DELAY/100)*{sw[9:0],1'b1}*(1+sw[11:10]*SCAN_DELAY_MULT);
if (wrong_move) begin // draw an X if there is a wrong move
if (~scan_enable &(counter<15*led_timer_big))
begin ja[2:0] <= {3'b010}[2:0];
ja[5:3] <= {3'b010}[2:0];end
else if (~scan_enable &(counter<30*led_timer_big))
begin ja[2:0] <= {3'b010}[2:0];
ja[5:3] <= {3'b100}[2:0]; end
else if (~scan_enable &(counter<45*led_timer_big))
begin ja[2:0] <= {3'b100}[2:0];
ja[5:3] <= {3'b010}[2:0]; end
else if (~scan_enable &(counter<60*led_timer_big))
begin ja[2:0] <= {3'b100}[2:0];
ja[5:3] <= {3'b100}[2:0]; end
else if (~scan_enable &(counter<79*led_timer_big))
begin ja[2:0] <= {3'b110}[2:0];
ja[5:3] <= {3'b001}[2:0]; end
end
else if(win_a) begin // if a wins then the first row
if (~scan_enable &(counter<15*led_timer_big))
begin ja[2:0] <= {3'b010}[2:0];
ja[5:3] <= {3'b000}[2:0]; end
else if (~scan_enable &(counter<30*led_timer_big))
begin ja[2:0] <= {3'b011}[2:0];
ja[5:3] <= {3'b000}[2:0]; end
else if (~scan_enable &(counter<45*led_timer_big))
begin ja[2:0] <= {3'b100}[2:0];
ja[5:3] <= {3'b000}[2:0]; end
else if (~scan_enable &(counter<60*led_timer_big))
begin ja[2:0] <= {3'b101}[2:0];
ja[5:3] <= {3'b000}[2:0]; end
else if (~scan_enable &(counter<79*led_timer_big))
begin ja[2:0] <= {3'b110}[2:0];
ja[5:3] <= {3'b000}[2:0]; end
end
else if(win_b) begin // if b wins
if (~scan_enable &(counter<15*led_timer_big))
begin ja[2:0] <= {3'b010}[2:0];
ja[5:3] <= {3'b111}[2:0];end
else if (!scan_enable && (counter<30*led_timer_big))
    begin ja[2:0] <= {3'b011}[2:0];
        ja[5:3] <= {3'b111}[2:0];
    end
else if (!scan_enable && (counter<45*led_timer_big))
    begin ja[2:0] <= {3'b100}[2:0];
        ja[5:3] <= {3'b111}[2:0];
    end
else if (!scan_enable && (counter<60*led_timer_big))
    begin ja[2:0] <= {3'b101}[2:0];
        ja[5:3] <= {3'b111}[2:0];
    end
else if (!scan_enable && (counter<79*led_timer_big))
    begin ja[2:0] <= {3'b110}[2:0];
        ja[5:3] <= {3'b111}[2:0];
    end
end

// Display required move
else begin // Display required move
    if (!scan_enable && (counter<40*led_timer_big))
        begin
            ja[2:0] <= from_to[2:0];
            ja[5:3] <= from_to[5:3];
        end
    else if (!scan_enable && (counter<80*led_timer_big))
        begin
            ja[2:0] <= from_to[8:6];
            ja[5:3] <= from_to[11:9];
        end
end

assign led16_b = ~jb[0];
assign led17_r = wrong_move;
assign led17_g = final_positions[7];
assign led17_b = final_positions[0];
assign led[15:0] = final_positions[63:48];

// SERIAL PART
logic clean;
logic old_clean;
logic sig_readdy2;

always_ff @(posedge clk_100mhz)
    begin
        old_clean <= clean; // for rising edge detection
        if (jc[0])
            from_to[2:0] <= jc[3:1];
        else if (~jc[0])
            from_to[5:3] <= jc[6:4];
    end

debounce my_deb(.clock_in(clk_100mhz),
                  .reset_in(btnd),
.noisy_in(btnc),
.clean_out(clean),
.ready(sig_readdy2));

serial_tx my_tx(.clk_in(clk_100mhz),
.rst_in(btnd),
.trigger_in(clean&~old_clean),
.val_in(computer_out),//should be data to computer
.data_out(ja[6]));

// Communication with the fsm
// input: final_positions
// output:

chess_fsm my_fsm(
.clk_100mhz(clk_100mhz),
.rst_in(btnd),
.player_first(1),
.board_state(final_positions),
.board_state_ready(board_state_ready),
.opponents_move(computer_out),
.from_to(from_to),
.invalid_state(wrong_move));

endmodule

module debounce (input reset_in, clock_in, noisy_in,
.output reg clean_out, reg ready);

reg [19:0] count;
reg new_input;

always_ff @(posedge clock_in)
if (reset_in) begin
    new_input <= noisy_in;
    clean_out <= noisy_in;
    count <= 0;
    ready <= 0; end
else if (noisy_in != new_input) begin
    new_input <= noisy_in;
    count <= 0;
    ready <= 0;
end
else if (count == 1000) begin
    clean_out <= new_input;
    ready <= 1;
end
else begin
    count <= count+1;
    ready <= 0;
end

end
endmodule

module serial_tx( input clk_in,
    input rst_in,
    input trigger_in,
    input [7:0] val_in,
    output logic data_out);

parameter DIVISOR = 868; // treat this like a constant!!

logic [9:0] shift_buffer; // 10 bits... interesting
logic [31:0] count;
logic [8:0] count2 = 0;
logic started = 0;
always @(posedge clk_in) begin
    if (trigger_in) begin
        count <= 0;
        started <= 1;
        count2 <= 0;
        shift_buffer <= {1'b1, val_in, 1'b0};
        data_out <= 1;
    end
    if (rst_in) begin
        count <= 0;
        shift_buffer <= {1'b1, val_in, 1'b0};
        started <= 0;
        count2 <= 0;
    end
    else if (started) begin
        if (count < DIVISOR)
            count <= count + 1;
        else if (count2 == 11)
            started <= 0;
        else begin
            count <= 0;
            count2 <= count2 + 1;
            data_out <= shift_buffer[0];
            shift_buffer <= [shift_buffer[0], shift_buffer[7:1]];
        end
    end
end
endmodule

// FSM MODULE

module chess_fsm( input clk_100mhz,
    input rst_in,
    input player_first,
    input [63:0] board_state,
    input board_state_ready,
output logic opponents_move,
output logic [7:0] from_to,
output logic invalid_state
);

//ai_comms ai_uart(.from_to_player(),.sent_val());

logic game_started = 1'b0;
logic orientation = ~player_first;
// Orientation 0:
// Black (opponent)
// White (player)

// Orientation 1:
// Black (player)
// White (opponent)
logic player_move;
logic [63:0] prev_board_state;
logic [3:0] chess_fsm [63:0];
// 0XXX Player 1XXX Opponent
// X000 Empty
// X001 Pawn
// X010 Rook
// X011 Knight
// X100 Bishop
// X101 Queen
// X110 King
// X111 Pawn -> XXX
logic [2:0] game_state;
// 0XX Player, 1XX Opponent
// X00 X's Move
// X01 Moving
// X10 Attacking
// X11 Castling
logic invalid_state = 1'b0;
logic [3:0] lifted1_type;
logic [5:0] lifted1_loc;
logic [3:0] lifted2_type;
logic [5:0] lifted2_loc;

logic [5:0] player_king_loc;
logic [5:0] opponent_king_loc;

function valid_move;
    input [63:0] prev_board_state; //Moving piece shouldn't exist
    input [2:0] piece_type;
    input [5:0] old_loc;
    input [5:0] new_loc;
    input attacking;
    begin
        reg [5:0] bigger_loc;
        }
reg [5:0] smaller_loc;
reg [5:0] difference;
reg moving_up;
reg [2:0] old_loc_col;
reg hits_right_in_1;
reg hits_left_in_1;
reg hits_right_in_2;
reg hits_left_in_2;
if (new_loc > old_loc) begin
  bigger_loc = new_loc;
  smaller_loc = old_loc;
  moving_up = 1'b1;
end else begin
  bigger_loc = old_loc;
  smaller_loc = new_loc;
  moving_up = 1'b0;
end
difference = bigger_loc - smaller_loc;
old_loc_col = old_loc % 8;
hits_right_in_1 = old_loc_col == 0;
hits_left_in_1 = old_loc_col == 7;
hits_right_in_2 = old_loc_col == 1;
hits_left_in_2 = old_loc_col == 6;
case (piece_type)
  3'b000: begin //Empty moves (complete)
    valid_move = 1'b0;
  end
  3'b001: begin //Pawn moves (complete)
    //Pawn moves up if vvv else, pawn moves down
    if (orientation ^ player_move) begin //player on bottom moving up, or opponent on bottom moving up
      if (~moving_up) begin
        valid_move = 1'b0;
      end else begin
        if (attacking) begin //Pawn is attacking
          if ((difference == 7 && ~hits_right_in_1) || (difference == 9 && ~hits_left_in_1)) begin
            valid_move = 1'b1;
          end else begin
            valid_move = 1'b0;
          end
        end else begin //Pawn is not attacking
          if (new_loc == old_loc + 8 || (old_loc < 16 && new_loc == old_loc + 16)) begin //Pawn moved correctly
            valid_move = 1'b1;
          end else begin //Pawn moved incorrectly
            valid_move = 1'b0;
          end
        end
      end
    end else begin //Pawn should be moving down -----
      if (moving_up) begin
        valid_move = 1'b0;
      end else begin
        //Pawn moves down if vvv else, pawn moves up
        if (orientation ^ player_move) begin //player on bottom moving down, or opponent on bottom moving down
          if (moving_up) begin
            valid_move = 1'b0;
          end else begin
            if (attacking) begin //Pawn is attacking
              if ((difference == -7 && hits_right_in_1) || (difference == -9 && hits_left_in_1)) begin
                valid_move = 1'b1;
              end else begin
                valid_move = 1'b0;
              end
            end else begin //Pawn is not attacking
              if (new_loc == old_loc - 8 || (old_loc > 63 && new_loc == old_loc - 16)) begin //Pawn moved correctly
                valid_move = 1'b1;
              end else begin //Pawn moved incorrectly
                valid_move = 1'b0;
              end
            end
          end
        end else begin //Pawn should be moving up -----
          if (~moving_up) begin
            valid_move = 1'b0;
          end else begin
            if (attacking) begin //Pawn is attacking
              if ((difference == -7 && hits_right_in_1) || (difference == -9 && hits_left_in_1)) begin
                valid_move = 1'b1;
              end else begin
                valid_move = 1'b0;
              end
            end else begin //Pawn is not attacking
              if (new_loc == old_loc - 8 || (old_loc > 63 && new_loc == old_loc - 16)) begin //Pawn moved correctly
                valid_move = 1'b1;
              end else begin //Pawn moved incorrectly
                valid_move = 1'b0;
              end
            end
          end
        end
      end
    end
end
if (attacking) begin //Pawn is attacking
  if ((difference == 7 && ~hits_left_in_1) || (difference == 9 && ~hits_right_in_1)) begin
    valid_move = 1'b1;
  end else begin
    valid_move = 1'b0;
  end
end else begin //Pawn is not attacking
  if (new_loc == old_loc - 8 || (old_loc >= 48 && new_loc == old_loc - 16)) begin //Pawn moved correctly
    valid_move = 1'b1;
  end else begin //Pawn moved incorrectly
    valid_move = 1'b0;
  end
end
end

3'b010: begin //Rook moves (complete)
  if (difference % 8 == 0) begin
    valid_move = 1'b1;
    if (moving_up) begin //Piece collision detection
      for (int i = 0; i < 8; i++) begin
        if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) > old_loc & 8*i + (new_loc % 8) < new_loc) begin
          valid_move = 1'b0;
        end
      end
    end else begin
      for (int i = 0; i < 8; i++) begin
        if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) < old_loc & 8*i + (new_loc % 8) > new_loc) begin
          valid_move = 1'b0;
        end
      end
    end else begin
      valid_move = 1'b1;
      if (new_loc > old_loc) begin
        for (int i = 0; i < 8; i++) begin //Piece collision detection
          if (prev_board_state[new_loc - (new_loc % 8) + i] & new_loc - (new_loc % 8) + i > old_loc &
              new_loc - (new_loc % 8) + i < new_loc) begin
            valid_move = 1'b0;
          end
        end
      end else begin
        for (int i = 0; i < 8; i++) begin //Piece collision detection
          if (prev_board_state[new_loc - (new_loc % 8) + i] & new_loc - (new_loc % 8) + i < old_loc &
              new_loc - (new_loc % 8) + i > new_loc) begin
            valid_move = 1'b0;
          end
        end
      end
    end
  end end
if (old_loc >= 8 & new_loc >= 8) begin
    if (old_loc < new_loc) begin
        old_loc = old_loc - (old_loc - (old_loc%8));
        new_loc = new_loc - (old_loc - (old_loc%8));
    end else begin
        old_loc = old_loc - (new_loc - (new_loc%8));
        new_loc = new_loc - (new_loc - (new_loc%8));
    end
    end
if (old_loc < 8 & new_loc < 8 & valid_move != 1'b0) begin
    valid_move = 1'b1;
end else begin
    valid_move = 1'b0;
end
end

3'b011: begin //Knight moves (complete)
    if (difference == 6 & ((moving_up & ~hits_right_in_2) | (~moving_up & ~hits_left_in_2)) |
       difference == 10 & ((moving_up & ~hits_left_in_2) | (~moving_up & ~hits_right_in_2)) |
       difference == 15 & ((moving_up & ~hits_right_in_1) | (~moving_up & ~hits_left_in_1)) |
       difference == 17 & ((moving_up & ~hits_left_in_1) | (~moving_up & ~hits_right_in_1)) begin
        valid_move = 1'b1;
    end else begin
        valid_move = 1'b0;
    end
end

3'b100: begin //Bishop moves (complete)
    valid_move = 1'b0; //Should be default value here
    for (int i = 1; i < 8 - old_loc_col; i++) begin //Check for validity on columns to left
        if ((new_loc == old_loc + 9*i) || (new_loc == old_loc - 7*i)) begin
            valid_move = 1'b1;
        end
    end
    for (int j = 1; j <= old_loc_col; j++) begin //Check for validity on columns to right
        if ((new_loc == old_loc - 9*j) || (new_loc == old_loc + 7*j)) begin
            valid_move = 1'b1;
        end
    end
    if (valid_move) begin //Piece collision detection
        if (moving_up) begin
            if (difference % 9 == 0) begin //Northwest
                for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
                    if (prev_board_state[old_loc + 9*k]) begin
                        valid_move = 1'b0;
                    end
                end
            end else begin //Northeast
                for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
                    if (prev_board_state[old_loc + 7*k]) begin
                        valid_move = 1'b0;
                    end
                end
            end
        end else begin //Southwest
            for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
                if (prev_board_state[old_loc + 7*k]) begin
                    valid_move = 1'b0;
                end
            end
        end
    end
end
end
end else begin
if (difference % 9 == 0) begin //Southeast
for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
    if (prev_board_state[old_loc - 9*k]) begin
        valid_move = 1'b0;
    end
end
end else begin //Southwest
for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
    if (prev_board_state[old_loc - 7*k]) begin
        valid_move = 1'b0;
    end
end
end
end
end
end
end
end
end
3'b101: begin //Queen moves (combine bishop and rook code)
    valid_move = 1'b0; //Should be default value here
    //vvv Bishop check vvv
    for (int i = 1; i < 8 - old_loc_col; i++) begin //Check for validity on columns to left
        if ((new_loc == old_loc + 9*i) || (new_loc == old_loc - 7*i)) begin
            valid_move = 1'b1;
        end
    end
    for (int j = 1; j <= old_loc_col; j++) begin //Check for validity on columns to right
        if ((new_loc == old_loc - 9*j) || (new_loc == old_loc + 7*j)) begin
            valid_move = 1'b1;
        end
    end
end
if (valid_move) begin //Piece collision detection
    if (moving_up) begin
        if (difference % 9 == 0) begin //Northwest
            for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
                if (prev_board_state[old_loc + 9*k]) begin
                    valid_move = 1'b0;
                end
            end
        end else begin //Northeast
            for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
                if (prev_board_state[old_loc + 7*k]) begin
                    valid_move = 1'b0;
                end
            end
        end
    end else begin
        if (difference % 9 == 0) begin //Southeast
            for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
                if (prev_board_state[old_loc - 9*k]) begin
                    valid_move = 1'b0;
                end
            end
        end else begin //Northwest
            for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
                if (prev_board_state[old_loc - 9*k]) begin
                    valid_move = 1'b0;
                end
            end
        end
    end
end
end
end else begin //Southwest
  for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
    if (prev_board_state[old_loc - 7*k]) begin
      valid_move = 1'b0;
    end
  end
end

//vvv Rook check vvv
if (~valid_move) begin //Failed to find valid bishop route
  if (difference % 8 == 0) begin
    valid_move = 1'b1;
    if (moving_up) begin //Piece collision detection
      for (int i = 0; i < 8; i++) begin
        if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) > old_loc & 8*i + (new_loc % 8) < new_loc) begin
          valid_move = 1'b0;
        end
      end
    end else begin
      for (int i = 0; i < 8; i++) begin
        if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) < old_loc & 8*i + (new_loc % 8) > new_loc) begin
          valid_move = 1'b0;
        end
      end
    end
  end else begin
    valid_move = 1'b1;
    if (new_loc > old_loc) begin //Piece collision detection
      for (int i = 0; i < 8; i++) begin
        if (prev_board_state[new_loc - (new_loc % 8) + i] & new_loc - (new_loc % 8) > old_loc & new_loc - (new_loc % 8) + i < new_loc) begin
          valid_move = 1'b0;
        end
      end
    end else begin
      for (int i = 0; i < 8; i++) begin //Piece collision detection
        if (prev_board_state[new_loc - (new_loc % 8) + i] & new_loc - (new_loc % 8) + i > old_loc & new_loc - (new_loc % 8) + i < new_loc) begin
          valid_move = 1'b0;
        end
      end
    end
  end
end else begin
  old_loc = old_loc - (old_loc - (old_loc%8));
  new_loc = new_loc - (old_loc - (old_loc%8));
end else begin
  old_loc = old_loc - (new_loc - (new_loc%8));
new_loc = new_loc - (new_loc - (new_loc%8));
end

if (old_loc < 8 & new_loc < 8 & valid_move != 1'b0) begin
  valid_move = 1'b1;
end else begin
  valid_move = 1'b0;
end
end

3'b110: begin //King moves (complete)
  if (difference == 1 & ((moving_up & ~hits_left_in_1) | (~moving_up & ~hits_right_in_1)) | //Moved to
  right (not hitting edge) or Moved to left (not hitting edge)
    difference == 7 & ((moving_up & ~hits_right_in_1) | (~moving_up & ~hits_left_in_1)) |
    difference == 8 |
    difference == 9 & ((moving_up & ~hits_left_in_1) | (~moving_up & ~hits_right_in_1)) ) begin
    valid_move = 1'b1;
  end else begin
    valid_move = 1'b0;
  end
end

3'b111: begin //Transitioning Pawn moves (Knight and Queen ability (will get upgraded by FSM after
first move))
  valid_move = 1'b0; //Should be default value here
  //vvv Bishop check vvv
  for (int i = 1; i < 8 - old_loc_col; i++) begin //Check for validity on columns to left
    if ((new_loc == old_loc + 9*i) || (new_loc == old_loc - 7*i)) begin
      valid_move = 1'b1;
    end
  end
  for (int j = 1; j <= old_loc_col; j++) begin //Check for validity on columns to right
    if ((new_loc == old_loc - 9*j) || (new_loc == old_loc + 7*j)) begin
      valid_move = 1'b1;
    end
  end
  if (valid_move) begin //Piece collision detection
    if (moving_up) begin
      if (difference % 9 == 0) begin //Northwest
        for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
          if (prev_board_state[old_loc + 9*k]) begin
            valid_move = 1'b0;
          end
        end
      end else begin //Northeast
        for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
          if (prev_board_state[old_loc + 7*k]) begin
            valid_move = 1'b0;
          end
        end
      end
    end else begin //Southeast
      for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
        if (prev_board_state[old_loc - 7*k]) begin
          valid_move = 1'b0;
        end
      end
    end
  end
end
if (difference % 9 == 0) begin //Southeast
    for (int k = 1; k <= old_loc_col; k++) begin //Check for validity on columns to right
        if (prev_board_state[old_loc - 9*k]) begin
            valid_move = 1'b0;
        end
    end
end else begin //Southwest
    for (int k = 1; k < 8 - old_loc_col; k++) begin //Check for validity on columns to left
        if (prev_board_state[old_loc - 7*k]) begin
            valid_move = 1'b0;
        end
    end
end
//vvv Rook check vvv
if (~valid_move) begin //Failed to find valid bishop route
    if (difference % 8 == 0) begin
        valid_move = 1'b1;
        if (moving_up) begin //Piece collision detection
            for (int i = 0; i < 8; i++) begin
                if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) > old_loc & 8*i + (new_loc % 8) < new_loc) begin
                    valid_move = 1'b0;
                end
            end
        end else begin
            for (int i = 0; i < 8; i++) begin
                if (prev_board_state[8*i + (new_loc % 8)] & 8*i + (new_loc % 8) > new_loc & 8*i + (new_loc % 8) < old_loc) begin
                    valid_move = 1'b0;
                end
            end
        end
    end else begin
        valid_move = 1'b1;
        if (new_loc > old_loc) begin
            for (int i = 0; i < 8; i++) begin //Piece collision detection
                if (prev_board_state[new_loc - (new_loc % 8) + i & new_loc - (new_loc % 8) + i > old_loc & new_loc - (new_loc % 8) + i < new_loc) begin
                    valid_move = 1'b0;
                end
            end
        end else begin
            for (int i = 0; i < 8; i++) begin
                if (prev_board_state[new_loc - (new_loc % 8) + i & new_loc - (new_loc % 8) + i < old_loc & new_loc - (new_loc % 8) + i > new_loc) begin
                    valid_move = 1'b0;
                end
            end
        end
    end
if (old_loc >= 8 & new_loc >= 8) begin
    if (old_loc < new_loc) begin
        old_loc = old_loc - (old_loc - (old_loc%8));
        new_loc = new_loc - (old_loc - (old_loc%8));
    end else begin
        old_loc = old_loc - (new_loc - (new_loc%8));
        new_loc = new_loc - (new_loc - (new_loc%8));
    end
end
if (old_loc < 8 & new_loc < 8 & valid_move != 1'b0) begin
    valid_move = 1'b1;
end else begin
    valid_move = 1'b0;
end
endcase
endfunction
always_ff @(posedge clk_100mhz) begin
    if (rst_in) begin
        game_started <= 1'b0;
    end else begin
        if (board_state_ready) begin //Only run game logic on valid board_states
            if (game_started) begin //usual logic fsm
                if (board_state != prev_board_state) begin //Something has changed on the board, and the board was valid prior
                    if (~invalid_state) begin
                        automatic reg [63:0] board_change = board_state ^ prev_board_state;
                        automatic reg [5:0] loc_change; //Where the change was detected
                        automatic reg update_state = 1'b1;
                        for (int i = 63; i > 0; i--) begin
                            if (board_change[i]) begin
                                loc_change = i;
                            end
                        end
                        case (game_state)
                            3'b000: begin //Player Move
                                if (board_state[loc_change]) begin //A piece was placed down (WE DON'T CURRENTLY SUPPORT PIECE PLACE PRIOR TO PROMOTION)
                                    invalid_state <= 1'b1;
                                    update_state = 1'b0;
                                valid_state <= 1'b1;
                            end
                        endcase
                    end
                end
            end
        end
    end
end
end else begin
    lifted1_type <= chess_fsm[loc_change]; //Which piece was lifted first
    lifted1_loc <= loc_change; //Where was it lifted from
    if (chess_fsm[loc_change][3]) begin //An opponent's piece was lifted
        game_state <= 3'b010; //Player is attacking
    end else begin //A player's piece was lifted
        game_state <= 3'b001; //Player is moving
    end
end

3'b001: begin //Player is moving
    if (loc_change == lifted1_loc) begin //Piece was placed back down in same spot. Go back to
        player's turn
        game_state <= 3'b000;
        lifted1_type <= 4'b0000;
    end else begin
        if (board_state[loc_change]) begin  //A piece was placed down (Player has made their move)
            if (valid_move(prev_board_state,lifted1_type[2:0],lifted1_loc,loc_change,1'b0)) begin
                //Virtually make the move (send game_state to valid_move w/ updated king_loc if
                //necessary)
                //Make sure the king is safe
                // If he is, write to fsm + forward to uart
                // If not, enter invalid state
                for (int i = 0; i < 64; i++) begin
                    if (chess_fsm[i][3]) begin //Opponent's piece
                        //If their exists an opponent piece which is capable of attacking the player's king,
                        this move is invalid
                        if (valid_move(board_state, chess_fsm[i][2:0], i, lifted1_type[2:0] == 3'b110 ?
                            loc_change : player_king_loc, 1'b1)) begin
                            invalid_state <= 1'b1;
                            update_state = 1'b0;
                        end
                    end
                end
            end else begin //Player made an invalid move
                invalid_state <= 1'b1;
                update_state = 1'b0;
            end
        end
        if (update_state) begin //Player is not in check, and their move was valid
            if (lifted1_type[2:0] == 3'b110) begin //Update the king position if needed
                player_king_loc <= loc_change;
            end
            chess_fsm[lifted1_loc] <= 4'b0000; //Clear out the original piece location
            chess_fsm[loc_change] <= lifted1_type; //Fill in the new piece location
            //Send stuff to UART
            game_state <= 3'b000; //Change back to 3'b100
            lifted1_type <= 4'b0000;
            player_move <= 1'b0;
        end
    end
end else begin //Player is either attacking or castling (piece was lifted)
if (chess_fsm[loc_change][3]) begin //They lifted an opponent piece, and are attacking
//Go ahead and put back in buffer 1 + switch
//With opponent in buffer 1, we can guarantee that buffer 2 will be empty if both pieces aren't lifted
lifted2_type <= lifted1_type;
lifted2_loc <= lifted1_loc;
lifted1_type <= chess_fsm[loc_change];
lifted1_loc <= loc_change;
game_state <= 4'b010;
end else begin //They lifted another of their own pieces, must be castling
automatic reg king_1_orig = (lifted1_type == 4'b0110) & ((lifted1_loc == 3 & orientation == 0) | (lifted1_loc == 59 & orientation == 1));
automatic reg king_2_orig = (chess_fsm[loc_change] == 4'b0110) & ((loc_change == 3 & orientation == 0) | (loc_change == 59 & orientation == 1));
automatic reg rook_1_orig = (lifted1_type == 4'b0010) & (((lifted1_loc == 0 | lifted1_loc == 7) & orientation == 0) | ((lifted1_loc == 56 | lifted1_loc == 63) & orientation == 1));
automatic reg rook_2_orig = (chess_fsm[loc_change] == 4'b0010) & (((loc_change == 0 | loc_change == 7) & orientation == 0) | ((loc_change == 56 | loc_change == 63) & orientation == 1));
if ((king_1_orig & rook_2_orig) | (king_2_orig & rook_1_orig)) begin
lifted2_type <= chess_fsm[loc_change];
lifted2_loc <= loc_change;
game_state <= 3'b011;
end else begin
invalid_state <= 1'b1;
update_state = 1'b0;
end
end
end
end
end
3'b010: begin //Player is attacking (Assumed that buffers are (opponent at 1 and player at 2)
if (lifted2_type == 4'b0000) begin //entered by lifting a black piece
if (game_state[loc_change]) begin //A piece was placed down (better be same piece in same spot)
if (loc_change == lifted1_loc) begin
lifted1_type <= 4'b0000;
game_state <= 3'b000;
end else begin
invalid_state <= 1'b1;
update_state = 1'b0;
end
end else begin //Another piece was lifted (better be their own piece)
if (chess_fsm[loc_change][3]) begin //Lifted another black piece
invalid_state <= 1'b1;
update_state = 1'b0;
end else begin //Lifted one of their own pieces
lifted2_type <= chess_fsm[loc_change];
lifted2_loc <= loc_change;
end
end
end else begin //entered by lifting a white piece then a black piece (black still in spot 1)
if (game_state[loc_change]) begin // A piece was placed down (better be in same spot as black)
  if (loc_change == lifted1_loc) begin // Correct
    if (valid_move(prev_board_state, lifted2_type[2:0], lifted2_loc, loc_change, 1'b1)) begin
      // Virtually make the move (send game_state to valid_move w/ updated king_loc if necessary)
      // Make sure the king is safe
      // If he is, write to fsm + forward to uart
      // If not, enter invalid state
      for (int i = 0; i < 64; i++) begin
        if (chess_fsm[i][3]) begin // Opponent's piece
          // If their exists an opponent piece which is capable of attacking the player's king, this move is invalid
          if (valid_move(board_state, chess_fsm[i][2:0], i, lifted2_type[2:0] == 3'b110 ? loc_change : player_king_loc, 1'b1)) begin
            invalid_state <= 1'b1;
            update_state = 1'b0;
            end
          end
        end
      end
      if (update_state) begin // Player is not in check, and their move was valid
        if (lifted2_type[2:0] == 3'b110) begin // Update the king position if needed
          player_king_loc <= loc_change;
        end
        chess_fsm[lifted2_loc] <= 4'b0000; // Clear out the original piece location
        chess_fsm[loc_change] <= lifted2_type; // Fill in the new piece location
        // Send stuff to UART
        game_state <= 3'b000; // Change back to 3'b100
        lifted1_type <= 4'b0000;
        lifted2_type <= 4'b0000;
        player_move <= 1'b0;
        end
      end else begin // Piece was moved improperly
        invalid_state <= 1'b1;
        update_state = 1'b0;
      end
    end
  else begin // Invalid
    invalid_state <= 1'b1;
    update_state = 1'b0;
  end
end else begin // Really... A third piece. invalidated!
  invalid_state <= 1'b1;
  update_state = 1'b0;
end
end else begin // Player is castling [Lift both pieces in question, then place them in the correct spots [atomic move]]
  automatic reg castle_left = (orientation == 0 & (lifted1_loc == 7 | lifted2_loc == 7)) |
  (orientation == 1 & (lifted1_loc == 63 | lifted2_loc == 63));
  if (lifted2_type == 4'b0000) begin // Player has already placed one piece down
if (game_state[loc_change]) begin //Second piece is placed down
  if ((orientation == 0 & castle_left & loc_change == 5) |
      (orientation == 0 & ~castle_left & loc_change == 1) |
      (orientation == 1 & castle_left & loc_change == 61) |
      (orientation == 1 & ~castle_left & loc_change == 57)) begin //King was placed
    if (valid_move(prev_board_state,4'b0010,lifted1_loc,loc_change,1'b0)) begin //Pretend
      king is rook for validity checking (can move more than 1)
      chess_fsm[lifted1_loc] <= 4'b0000;
      chess_fsm[loc_change] <= 4'b0110;
      player_move <= 1'b0;
      game_state <= 3'b000; //Change back to 3'b100
      lifted1_type <= 4'b0000;
    end else begin
      invalid_state <= 1'b1;
      update_state = 1'b0;
    end
  end else begin //Rook was placed
    if (valid_move(prev_board_state,lifted1_type[2:0],lifted1_loc,loc_change,1'b0)) begin
      chess_fsm[lifted1_loc] <= 4'b0000;
      chess_fsm[loc_change] <= 4'b0010;
      player_move <= 1'b0;
      game_state <= 3'b000; //Change back to 3'b100
      lifted1_type <= 4'b0000;
    end else begin
      invalid_state <= 1'b1;
      update_state = 1'b0;
    end
  end else begin //Placed them down in an invalid spot
    invalid_state <= 1'b1;
    update_state = 1'b0;
end
end else begin //Atomic operation. Once castling has begun, no re-lifting
  invalid_state <= 1'b1;
  update_state = 1'b0;
end
end else begin //Player has lifted both pieces at this point
  if (game_state[loc_change]) begin //A piece was placed
    if (loc_change == lifted1_loc | loc_change == lifted2_loc) begin //Piece was put back where
      it started
        if (loc_change == lifted1_loc) begin
          lifted1_loc <= lifted2_loc;
          lifted1_type <= lifted2_type;
        end
        lifted2_type <= 4'b0000;
        game_state <= 3'b001;
      end
    end
    if (loc_change == lifted2_loc) begin
      lifted2_loc <= lifted1_loc;
      lifted2_type <= lifted1_type;
    end
  end
end
end else begin // First piece was placed down
  if ((orientation == 0 & castle_left & loc_change == 5) |
      (orientation == 0 & ~castle_left & loc_change == 1) |
      (orientation == 1 & castle_left & loc_change == 61) |
      (orientation == 1 & ~castle_left & loc_change == 57)) begin // King was placed down
    first (Pretend is rook for validity)
    if (lifted1_type == 4'b0110) begin // King is in 1 slot
      if (valid_move(prev_board_state,4'b0010,lifted1_loc,loc_change,1'b0)) begin
        chess_fsm[lifted1_loc] <= 4'b0000;
        chess_fsm[loc_change] <= 4'b0010;
      end else begin
        invalid_state <= 1'b1;
        update_state = 1'b0;
      end
      lifted1_loc <= lifted2_loc;
      lifted1_type <= lifted2_type;
    end else begin // King is in 2 slot
      if (valid_move(prev_board_state,4'b0010,lifted2_loc,loc_change,1'b0)) begin
        chess_fsm[lifted2_loc] <= 4'b0000;
        chess_fsm[loc_change] <= 4'b0010;
      end else begin
        invalid_state <= 1'b1;
        update_state = 1'b0;
      end
    end
    lifted2_type <= 4'b0000;
  end else begin
    if ((orientation == 0 & castle_left & loc_change == 4) |
        (orientation == 0 & ~castle_left & loc_change == 2) |
        (orientation == 1 & castle_left & loc_change == 60) |
        (orientation == 1 & ~castle_left & loc_change == 58)) begin // Rook was placed down
      first
      if (lifted1_type == 4'b0010) begin // Rook is in 1 slot
        if (valid_move(prev_board_state,lifted1_type[2:0],lifted1_loc,loc_change,1'b0))
        begin
          chess_fsm[lifted1_loc] <= 4'b0000;
          chess_fsm[loc_change] <= 4'b0010;
        end else begin
          invalid_state <= 1'b1;
          update_state = 1'b0;
        end
        lifted1_loc <= lifted2_loc;
        lifted1_type <= lifted2_type;
      end else begin // Rook is in 2 slot
        if (valid_move(prev_board_state,lifted2_type[2:0],lifted2_loc,loc_change,1'b0))
        begin
          chess_fsm[lifted2_loc] <= 4'b0000;
          chess_fsm[loc_change] <= 4'b0010;
        end else begin
          invalid_state <= 1'b1;
        end
      end
    end
  end
update_state = 1'b0;
end
end
lifted2_type <= 4'b0000;
end else begin //Placed them down in an invalid spot
invalid_state <= 1'b1;
end
end
update_state = 1'b0; //Makes checking for move validity much easier
end
end else begin //Picked up a third piece. XD
invalid_state <= 1'b1;
update_state = 1'b0;
end
end
end
3'b100: begin //Opponent's turn
end
3'b101: begin //Opponent is moving
end
3'b110: begin //Opponent is attacking
end
3'b111: begin //Opponent is castling
end
case
endcase
if (update_state) begin
prev_board_state <= board_state;
end
end
end else begin //States do match
if (invalid_state) begin //Since we don't update prev_state when we detect invalid state, this means
we have returned to the valid state
invalid_state <= 1'b0;
end
end else begin //Game has not started yet (reset has been pushed)
automatic reg royal_rows_present = &board_state[63:56] & &board_state[7:0]; //The top/bottom rows
are fully occupied
automatic reg pawn_rows_present = &board_state[55:48] & &board_state[15:8]; //The pawn rows are
fully occupied
automatic reg other_rows_present = |board_state[47:16]; //There's something in the middle of the
board
if (royal_rows_present & pawn_rows_present & ~other_rows_present) begin //Valid starting state
game_started <= 1'b1;
prev_board_state <= board_state;
if (player_first) begin
game_state <= 3'b000; //Player's Move
chess_fsm <=
{{4'b1010},{4'b1011},{4'b1100},{4'b1101},{4'b1110},{4'b1111},{4'b1010},{4'b1100},
{4'b1001},{4'b1000},{4'b1001},{4'b1000},{4'b1001},{4'b1000},{4'b1001},{4'b1000},
{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},
{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},{4'b0000},
module ai_uart(
    input [7:0] from_to_player,
    input send_val,
    output [7:0] from_to_ai,
    output rcv_val
);

endmodule