ACTIVITY 6B: Finite State Machine (Ping Pong Games)

Jason Harvey Lorenzo, April 15, 2024
ENGG 125.03, Department of Electronics, Computer, and Communications Engineering
School of Science and Engineering, Ateneo de Manila University
Quezon City, 1108, Philippines
jason.lorenzo@obf.ateneo.edu

***Abstract*—This paper presents a Verilog HDL implementation of a Finite State Machine (FSM) for a simple ping pong game. The game utilizes switches and push buttons for player interaction, with visual feedback provided by virtual LEDs. The FSM is divided into sequential and combinational parts, with the former managing timing and parameter updates and the latter handling state transitions and outputs. Notably, the code incorporates special functionality for paddle hits at specific positions, adjusting ball speed accordingly. Simulation results confirm the expected behavior of the game, including scoring, speed changes, and resets, validating the effectiveness of the implemented FSM. Despite Quartus not extracting a state machine due to the code's complexity, the output circuit successfully controls game logic and outputs.**

***Index Terms*—Verilog HDL, finite state machine (FSM), ping pong game**

I. Introduction

This project involves the creation of a Finite State Machine (FSM) for a simple ping pong game using Verilog HDL. The game's states will be visually represented by virtual LEDs (L0 to L9), with each lit LED indicating the position of the ping pong ball. Players 1 and 2 control their paddles with switches PB0 and PB1, respectively. These switches are configured to be edge-triggered to prevent continuous activation upon long presses, ensuring that the ball can only be hit during the switch's edge change. Score1 and Score2 registers will track the players' scores, with a maximum score of 7, exceeding resets the game to 0-0. Additionally, SW0 can reset the game and the scores. SW1 and SW2 control the ball's speed, with the selected speed determining the initial speed of the ball for each turn. See Table I for the speed settings.

TABLE I
Speed Settings

SW1	SW2	Speed
0	0	1 = Slow (400ms / state)
0	1	2 = Normal (200ms / state)
1	0	3 = Fast (100ms / state)
1	1	4 = Fastest (50ms / state)

Players have the opportunity to hit the "ball" at specific positions, namely pos1 and pos2, represented by L0 and L1 (for player1), and L9 and L8 (for player2), respectively. When a player successfully hits the ball back at pos1, the ball's speed remains at the set speed. However, if the player hits the ball back at pos2, the ball's speed increases by one unit, capped at the maximum setting (e.g., if the setting is 4, the speed remains at 4).

II. HDL Code

Listing 1 shows an HDL code implementation of the FSM.

Listing 1. Ping Pong Game FSM HDL Code

```
module pingpong(
  input CLK, SW0, SW1, SW2, PB0, PB1,
  output wire [9:0] LEDs,
  output reg [8:0] maxT,
  output reg [2:0] Score1, Score2
);
parameter IDLE=10'b0000000000,
                    L0=10'b1000000000,
                    L1=10'b0100000000,
                    L2=10'b0010000000,
                    L3=10'b0001000000,
                    L4=10'b0000100000,
                    L5=10'b0000010000,
                    L6=10'b0000001000,
                    L7=10'b0000000100,
                    L8=10'b0000000010,
                    L9=10'b0000000001,
                    maxScore=3'd7;

edgeDetect peED(CLK, PB0, PB1,
                RPE0, RPE1);

wire T, RPE0, RPE1;
wire [8:0] speed_select;
reg [9:0] state, next_state;
reg [8:0] count, next_maxT;
reg next_dir, state_dir, dir;
reg [2:0] next_Score1, next_Score2;

assign T = (count == maxT);
assign LEDs = state; // moore
assign speed_select = SW1 ? ( SW2 ? 9'd50
      : 9'd100) : (SW2 ? 9'd200 : 9'd400);
assign gameOver = (Score1 == 3'd7) |
```

```
                (Score2 == 3'd7);

initial begin
  state <= L0;
  next_state <= L0;
  maxT <= 9'd400;
  next_maxT <= 9'd400;
  count <= 9'd0;
  dir <= 1'b1; // going right
  next_dir <= 1'b1;
  Score1 <= 3'd0;
  Score2 <= 3'd0;
  next_Score1 <= 3'd0;
  next_Score2 <= 3'd0;
end

always @(posedge CLK or posedge SW0)
begin
  if (SW0) begin
    count <= 9'd0;
    dir <= 1'b1;
    maxT <= speed_select;
  end
  else begin
    count <= T ? 9'd0 : (count + 1'b1);
    dir <= next_dir;
    maxT <= next_maxT;
  end
end

always @(posedge T or posedge SW0)
begin
  if (SW0) begin
    Score1 <= 3'd0;
    Score2 <= 3'd0;
    state <= next_state;
    state_dir <= 1'b1;
  end
  else begin
    Score1 <= gameOver ? 3'd0 :
              next_Score1;
    Score2 <= gameOver ? 3'd0 :
              next_Score2;
    state <= next_state;
    state_dir <= next_dir;
  end
end

always @(state or RPE0 or RPE1 or
  dir or Score1 or Score2 or maxT
  or SW1 or SW2)
begin
    case(state)
      IDLE: begin
        next_state <= dir ? L0 : L9;
        next_dir <= ~state_dir;
        next_maxT <= speed_select;
        next_Score1 <= dir ?
          Score1 : (Score1 + 1'b1);
        next_Score2 <= dir ?
          (Score2 + 1'b1) : Score2;
      end
      L0: begin
        next_state <= dir ? L1 : IDLE;
        next_dir <= RPE0 ? 1'b1 : dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L1: begin
        next_state <= dir ? L2 : L0;
        next_dir <= RPE0 ? 1'b1 : dir;
        if (RPE0) next_maxT <=
          (maxT == 6'd50) ? maxT :
          {1'b0, maxT[8:1]}; // halves
        else next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L2: begin
        next_state <= dir ? L3 : L1;
        next_dir <= dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L3: begin
        next_state <= dir ? L4 : L2;
        next_dir <= dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L4: begin
        next_state <= dir ? L5 : L3;
        next_dir <= dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L5: begin
        next_state <= dir ? L6 : L4;
        next_dir <= dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L6: begin
        next_state <= dir ? L7 : L5;
        next_dir <= dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
```

```
        next_Score2 <= Score2;
      end
      L7: begin
        next_state <= dir ? L8 : L6;
        next_dir <= dir;
            next_maxT <= maxT;
            next_Score1 <= Score1;
        next_Score2 <= Score2;
          end
      L8: begin
        next_state <= dir ? L9 : L7;
        next_dir <= RPE1 ? 1'b0 : dir;
        if (RPE1) next_maxT <=
          (maxT == 6'd50) ? maxT :
          {1'b0, maxT[8:1]}; // halves
        else next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
      L9: begin
        next_state <= dir ? IDLE : L8;
        next_dir <= RPE1 ? 1'b0 : dir;
        next_maxT <= maxT;
        next_Score1 <= Score1;
        next_Score2 <= Score2;
      end
    endcase
end
endmodule

module edgeDetect(
  input CLK, PB0, PB1,
  output reg RPE0, RPE1
);

reg RPB0, RPB1;

always @(posedge CLK)
begin
  RPB0 <= PB0;
  RPB1 <= PB1;
  RPE0 <= PB0 & ~RPB0;
  RPE1 <= PB1 & ~RPB1;
end

endmodule
```

This code implements the logic for the ping pong game, where the players use switches SW0, SW1, and SW2 to control game parameters and push buttons (PB0 and PB1) to interact with the game. This FSM can be divided into its sequential and combinational parts. Note that dir refers to what direction the ball is going to. A dir of 1 indicates that the ball is going from left to right (from player1 to player2), and vice versa. Moreover, the edgeDetect module detects the edges of PB0 and PB1 to ensure that actions are triggered only on button presses, outputting the processed signal in registers RPB0 and RPB1, respectively.

The sequential part consists of two always blocks, one dependent on CLK and the other on T. CLK is a set input with a constant period while T can change in period as defined by the speed settings. To determine when T activates, a counter was used to count up to the number of cycles necessary for the CLK to achieve maxT (maximum count value to trigger T, implementing the speed level). A lower maxT entails that the count will reach it faster, triggering T more frequently. For this reason, this always block also controls the changing of value of maxT, updating it as soon as sequentially possible. Similarly, dir is also updated within this block. T's always block is where the state and scores registers are changed. Unlike the previous registers, these are better changed only when necessary (to produce output).

The combinational part consists of one always block sensitive on the change of states and user inputs. Depending on the current state, the inputs and intermediary registers are treated accordingly. All states set the next_state to the corresponding neighboring state depending on dir. For states L3 to L7, these are the only parameters that can be changed. The pos1 and pos2 states (L0 and L1, L8 and L9) as discussed in the introduction have more special functions, however. When a positive edge of paddle input (via RPB0 or RPB1) is detected in these states, dir is immediately switched (hitting the ball back in effect). When the ball is hit on L0 or L9, the speed (in terms of maxT) is not changed, however on L1 or L8, the speed is halved by right bit-shifting maxT (capped at maxT=50). IDLE is a crucial state as it can only be entered when a player fails to hit the ball, resulting to it going out of bounds (all LEDs at 0). This means the end of a turn and the start of another. When ending a turn, it increments the score of the scoring player by 1 which it identifies using dir. If the ball was going right when it went out of bounds, then player1 scored and vice versa. When starting the turn, it sets the non-scoring player to be the next server (or player1 when the game just started or reset). Then, depending on SW1 or SW2, speed will be set according to the specifications in Table I.

III. Output Circuit

Fig. 1 shows the output circuit of the code above. Quartus didn't detect any state machine from the code likely due to the use of multiple always blocks in the sequential part to manipulate the parameters. The use of multiple intermediary parameters also likely increased the complexity of the circuit. The approach used in this paper necessitated multiple always blocks, and with it, the intermediary parameters to accommodate the variable speed functionality.

Since Quartus did not extract any state machine from the code, much less abstracting one, the flipflops holding the states can be seen on the left-hand side. The rest of the circuit controls the next states (sequential) and drawing out the outputs from each state. Note that the approach in this paper used a Moore machine with the outputs depending on the

Fig. 1. Ping Pong Game Implementation RTL Netlist

states alone, specifically, the states themselves are the output (LEDs).

IV. Simulation

A. Paddle Hits and Changing Speed

Fig. 2 shows the simulation of paddle hits within a sample game. As seen, when the edge of PB1 is detected on state L8 when the ball is going right, the speed doubles by halving maxT from 400 to 200. The same goes for when the edge of PB0 is detected on state L1 when the ball is going left. As seen, maxT halved from 200 to 100. The states' period shrinks as maxT halves as expected.

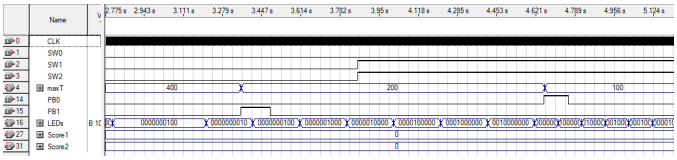

Fig. 2. Pos2 Paddle Hits Simulation Waveforms

Fig. 3 shows that when the ball is hit at pos1 like in state L9, ball speed remain constant. It also shows how the paddle input is only registered when the positive edge of that input lies on the concerned state. In this case, since PB0 (player1's paddle button) was pushed on state L3 (0010000000), it didn't hit the ball back despite pressing the button for longer. This eventually led to an IDLE state, and the incrementing of player2's score.

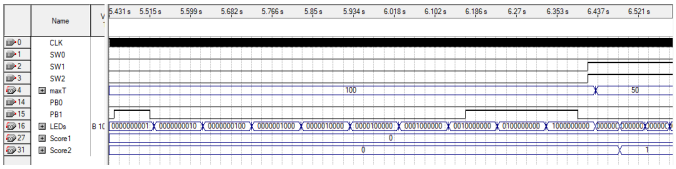

Fig. 3. Miscellaneous Paddle Hits Simulation Waveforms

Note as well that maxT was rightfully set to 50 during the IDLE state as SW1 and SW2 are both HIGH. Fig. 4 establishes this further as player2 scores, entering an IDLE state, only this time SW1=1 and SW2=0. This calls for a fast speed (maxT=100).

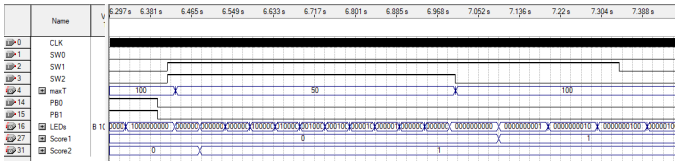

Fig. 4. SW1 and SW2 Effects

B. Scoring and Resets

Zooming further out, Fig. 5 shows that the scores appropriately increments when a player scores. They also reset when either of the players reaches 7 points.

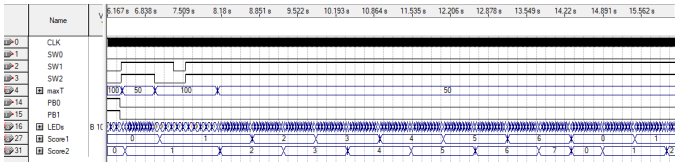

Fig. 5. No Hits, Very Fast Speed Simulation Waveforms

All these simulation results are in line with the functionalities defined.