

EECE 235 Project 4 Whack A Mole

Izatt, Avery

[YouTube Link](#)

Introduction:

The game uses an 8 by 8 LED matrix to show a single mole as a lit dot and four external push buttons as the player controls. The FPGA runs a finite state machine that handles the start screen, mole movement, scoring, timer, lives, and the game over state. This lets me practice using state machines, random mole positions, and hardware control of the LED matrix and seven segment displays.

How to play:

When the board is at idle the game shows a START message on the seven segment displays and all three lives are lit on LEDs 0, 1, and 2. To begin the game the player turns on the start switch (SW0). A single mole dot will appear somewhere on the 8 by 8 matrix. The matrix is divided into four quadrants, one for each push button. When the mole appears the player must press the button that matches the quadrant where the mole is lit. If the player hits the correct button the score increases by one and the mole moves to a new random location. If the player presses the wrong button or runs out of time, they lose one life. The lives LEDs turn off one by one as the player makes mistakes. When all three lives are gone the game enters the game over state and shows LOSE on the seven segment display. The high score can be viewed using the stats switch. To play again the player turns the start switch off and then on to return to the START screen and begin a new game.

Code:

```

1 module Proj4WhackAMole(
2
3     input  MAX10_CLK1_50,
4     input  [9:0] SW,
5     inout  [35:0] GPIO,
6
7     output [7:0] HEX0,
8     output [7:0] HEX1,
9     output [7:0] HEX2,
10    output [7:0] HEX3,
11    output [7:0] HEX4,
12    output [7:0] HEX5,
13    output [9:0] LEDR
14 );
15
16 // =====
17 // Clock Divider 100 kHz for matrix scan
18 // =====
19 wire clk100k;
20 ClockDivider100k div1(
21     .clk_in(MAX10_CLK1_50),
22     .clk_out(clk100k)
23 );
24
25 // Row scan counter for LED matrix (fast)
26 reg [2:0] row_sel = 0;
27 always @(posedge clk100k)
28     row_sel <= row_sel + 1;
29
30 // =====
31 // Slow game clock for FSM (about 100 Hz)
32 // =====
33 wire clk_game;
34
35 SlowGameClock game_div(
36     .clk_in(clk100k),    // 100 kHz in
37     .clk_out(clk_game)  // about 100 Hz out
38 );
39
40 // =====
41 // External buttons (active low)
42 // =====
43 wire smash_TL = ~GPIO[29];
44 wire smash_TR = ~GPIO[31];
45 wire smash_BL = ~GPIO[33];
46 wire smash_BR = ~GPIO[35];
47
48 wire smash_any = smash_TL | smash_TR | smash_BL | smash_BR;
49
50 // encode zone
51 // 00 top left, 01 top right, 10 bottom left, 11 bottom right
52 wire [1:0] smash_zone =
53     smash_TL ? 2'b00 :
54     smash_TR ? 2'b01 :
55     smash_BL ? 2'b10 :
56     smash_BR ? 2'b11 :
57     2'b11; // default when no button
58
59

```

```

58
59 // =====
60 // Switches
61 // =====
62 wire start      = SW[0]; // start game
63 wire reset      = SW[1]; // reset game
64 wire stats      = SW[2]; // show stats after lose
65 wire difficulty = SW[3]; // 0 easy, 1 hard
66
67 // =====
68 // FSM outputs
69 // =====
70 wire [1:0] mole_row_group;
71 wire [1:0] mole_col_group;
72
73 wire [7:0] score;
74 wire [7:0] timer_val; // hundredths of a second
75 wire [7:0] high_score;
76 wire [2:0] lives; // three lives: 3,2,1,0
77 wire      show_mole;
78
79 // =====
80 // FSM
81 // =====
82 WhackFSM_QSMASH fsm(
83     .clk(clk_game), // slow game clock
84     .start(start),
85     .reset(reset),
86     .smash_zone(smash_zone),
87     .smash_any(smash_any),
88     .stats(stats),
89     .difficulty(difficulty),
90     .mole_row_group(mole_row_group),
91     .mole_col_group(mole_col_group),
92     .score(score),
93     .timer_out(timer_val),
94     .high_score(high_score),
95     .lives(lives),
96     .show_mole(show_mole)
97 );
98
99 // =====
100 // Matrix Driver (single dot mole)
101 // =====
102 wire [7:0] row_drive;
103 wire [7:0] col_drive;
104
105 MatrixDriver2x2 matrix(
106     .row_sel(row_sel),
107     .mole_row_group(mole_row_group),
108     .mole_col_group(mole_col_group),
109     .show(show_mole),
110     .row_out(row_drive),
111     .col_out(col_drive)
112 );
113
114 // =====
115 // GPTO OUTPUT MAPPING

```

```

112 ];
113
114 // =====
115 // GPIO OUTPUT MAPPING
116 // =====
117 // Rows needed inversion at the pins for your panel
118
119 assign GPIO[0] = ~row_drive[0];
120 assign GPIO[2] = ~row_drive[1];
121 assign GPIO[4] = ~row_drive[2];
122 assign GPIO[6] = ~row_drive[3];
123 assign GPIO[8] = ~row_drive[4];
124 assign GPIO[10] = ~row_drive[5];
125 assign GPIO[12] = ~row_drive[6];
126 assign GPIO[14] = ~row_drive[7];
127
128 // Columns are active low as driven
129 assign GPIO[16] = col_drive[0];
130 assign GPIO[18] = col_drive[1];
131 assign GPIO[20] = col_drive[2];
132 assign GPIO[22] = col_drive[3];
133 assign GPIO[24] = col_drive[4];
134 assign GPIO[26] = col_drive[5];
135 assign GPIO[28] = col_drive[6];
136 assign GPIO[30] = col_drive[7];
137
138 // =====
139 // Lives and HEX displays
140 // =====
141
142 // lives is 3,2,1,0 and we want three LEDs in a row
143 // LED0 = life 1, LED1 = life 2, LED2 = life 3
144
145 assign LEDR[0] = (lives >= 1);
146 assign LEDR[1] = (lives >= 2);
147 assign LEDR[2] = (lives >= 3);
148
149 // other LEDs unused here
150 assign LEDR[9:3] = 7'b0000000;
151
152 // -----
153 // Seven segment digit selection
154 // -----
155 reg [3:0] d0, d1, d2, d3, d4, d5;
156 reg [7:0] sec;
157 reg [7:0] hund;
158
159 // idle when fresh: three lives, no score, no mole
160 wire in_idle = (lives == 3 && score == 0 && show_mole == 0);
161 // game over only when out of lives
162 wire game_over = (lives == 0);
163
164 // turn decimal point on except when showing stats
165 wire show_dp = !(game_over && stats);
166
167 always @(*) begin
168     // default numeric display
169     // score on HEX0 and HEX1

```

```

166
167 always @(*) begin
168     // default numeric display
169     // score on HEX5 and HEX4
170     d5 = score / 10;
171     d4 = score % 10;
172
173     // timer_val is in hundredths of a second at 100 Hz
174     sec = timer_val / 8'd100; // whole seconds
175     hund = timer_val % 8'd100; // hundredths
176
177     // time as SS.HH on HEX3..HEX0
178     d3 = sec / 10; // tens of seconds
179     d2 = sec % 10; // ones of seconds
180     d1 = hund / 10; // tens of hundredths
181     d0 = hund % 10; // ones of hundredths
182
183     // idle screen: show "START"
184     if (in_idle) begin
185         d5 = 4'hF; // blank
186         d4 = 4'd5; // S (digit 5)
187         d3 = 4'hC; // T
188         d2 = 4'hA; // A
189         d1 = 4'hD; // R
190         d0 = 4'hC; // T
191     end
192     // game over and stats switch off: show LOSE
193     else if (game_over && !stats) begin
194         d5 = 4'hF; // blank
195         d4 = 4'h8; // L
196         d3 = 4'd0; // O
197         d2 = 4'd5; // S
198         d1 = 4'hE; // E
199         d0 = 4'hF; // blank
200     end
201     // game over and stats switch on: show high score only
202     else if (game_over && stats) begin
203         d5 = 4'hF; // blank
204         d4 = 4'hF; // blank
205         d3 = 4'hF; // blank
206         d2 = 4'hF; // blank
207         d1 = high_score / 10; // tens
208         d0 = high_score % 10; // ones
209     end
210 end
211
212 // -----
213 // Drive the seven segment displays
214 // -----
215 wire [7:0] h0, h1, h2, h3, h4, h5;
216
217 SevenSeg s0(d0, h0);
218 SevenSeg s1(d1, h1);
219 SevenSeg s2(d2, h2);
220 SevenSeg s3(d3, h3);
221 SevenSeg s4(d4, h4);
222 SevenSeg s5(d5, h5);
223

```

```

220 SevenSeg s3(d3, h3);
221 SevenSeg s4(d4, h4);
222 SevenSeg s5(d5, h5);
223
224 // decimal point between seconds and hundredths for time
225 // HEX3 HEX2 . HEX1 HEX0 is "SS.HH" when show_dp is true
226 assign HEX0 = h0;
227 assign HEX1 = h1;
228 assign HEX2 = show_dp ? (h2 & 8'b01111111) : h2; // dp on except in stats
229 assign HEX3 = h3;
230 assign HEX4 = h4;
231 assign HEX5 = h5;
232
233 endmodule
234
235
236 // =====
237 // QUADRANT WHACK FSM (no win cap, infinite score)
238 // =====
239 module WhackFSM_QSMASH(
240     input clk,
241     input start,
242     input reset,
243     input [1:0] smash_zone,
244     input smash_any,
245     input stats,
246     input difficulty,
247
248     output reg [1:0] mole_row_group,
249     output reg [1:0] mole_col_group,
250
251     output reg [7:0] score,
252     output reg [7:0] timer_out, // hundredths
253     output reg [7:0] high_score,
254     output reg [2:0] lives, // 3,2,1,0
255     output reg show_mole
256 );
257
258 reg [2:0] state = 3'd0;
259
260 localparam IDLE = 3'd0;
261 localparam LIGHT_MOLE = 3'd1;
262 localparam WAIT_INPUT = 3'd2;
263 localparam CHECK = 3'd3;
264 localparam SCORE_UPDATE = 3'd4;
265 localparam NEXT_ROUND = 3'd5;
266 localparam GAME_OVER = 3'd6;
267
268 reg [15:0] timer;
269 reg [15:0] allowed_time = 16'd200; // about 2 seconds at 100 Hz
270
271 // LFSR for mole position
272 reg [7:0] lfsr = 8'hA3;
273
274 // Quadrant decode from group bits
275 wire [1:0] mole_zone =
276     (mole_row_group < 2 && mole_col_group < 2) ? 2'b00 :
277     (mole_row_group < 2 && mole_col_group > 1) ? 2'b01 :

```

```

274 // Quadrant decode from group bits
275 wire [1:0] mole_zone =
276     (mole_row_group < 2 && mole_col_group < 2) ? 2'b00 :
277     (mole_row_group < 2 && mole_col_group > 1) ? 2'b01 :
278     (mole_row_group > 1 && mole_col_group < 2) ? 2'b10 :
279     2'b11;
280
281 // edge detection for smash_any
282 reg prev_smash_any = 0;
283 wire smash_edge = smash_any & ~prev_smash_any;
284
285 // edge detection for start (to restart after game over)
286 reg prev_start = 0;
287 wire start_edge = start & ~prev_start;
288
289 reg [7:0] new_score;
290
291 always @(posedge clk) begin
292     prev_smash_any <= smash_any;
293     prev_start <= start;
294
295     if (reset) begin
296         state <= IDLE;
297         score <= 0;
298         lives <= 3; // three lives
299         show_mole <= 0;
300         timer_out <= 0;
301         allowed_time <= difficulty ? 16'd120 : 16'd200;
302         lfsr <= 8'hA3;
303         high_score <= 0;
304     end else begin
305
306         case(state)
307
308             IDLE: begin
309                 show_mole <= 0;
310                 timer_out <= 0;
311                 allowed_time <= difficulty ? 16'd120 : 16'd200;
312                 score <= 0;
313                 lives <= 3; // full lives on each new game
314                 if (start)
315                     state <= LIGHT_MOLE;
316             end
317
318             LIGHT_MOLE: begin
319                 lfsr <= {lfsr[6:0], lfsr[7] ^ lfsr[5]};
320
321                 mole_row_group <= lfsr[1:0];
322                 mole_col_group <= lfsr[3:2];
323
324                 show_mole <= 1;
325                 timer <= allowed_time;
326                 timer_out <= 0;
327                 state <= WAIT_INPUT;
328             end
329
330             WAIT_INPUT: begin
331                 if (timer == 0)

```

```

328 end
329
330 WAIT_INPUT: begin
331     if (timer != 0)
332         timer <= timer - 1;
333
334         timer_out <= timer_out + 1;
335
336     if (smash_edge) begin
337         if (smash_zone == mole_zone)
338             state <= SCORE_UPDATE;
339         else
340             state <= CHECK; // wrong button press
341     end else if (timer == 0) begin
342         state <= CHECK; // timeout also counts as a miss
343     end
344 end
345
346 CHECK: begin
347     // miss or timeout: lose one life
348     if (lives <= 1) begin
349         lives <= 0;
350         show_mole <= 0;
351         state <= GAME_OVER;
352     end else begin
353         lives <= lives - 1;
354         state <= NEXT_ROUND;
355     end
356 end
357
358 SCORE_UPDATE: begin
359     new_score = score + 1;
360     score <= new_score;
361
362     if (new_score > high_score)
363         high_score <= new_score;
364
365     // speed up a little, but keep a reasonable minimum
366     if (allowed_time > 16'd60)
367         allowed_time <= allowed_time - 16'd10;
368
369     state <= NEXT_ROUND;
370 end
371
372 NEXT_ROUND: begin
373     show_mole <= 0;
374     state <= LIGHT_MOLE;
375 end
376
377 GAME_OVER: begin
378     show_mole <= 0;
379     // Stay in GAME_OVER until the user toggles start OFF then ON.
380     if (start_edge) begin
381         state <= IDLE;
382     end
383 end
384
385 default: begin

```

```

385         default: begin
386             state <= IDLE;
387         end
388     endcase
389 end
390 end
391 end
392
393 endmodule
394
395
396 // =====
397 // MATRIX DRIVER, SINGLE DOT
398 // =====
399 module MatrixDriver2x2(
400     input  [2:0] row_sel,
401     input  [1:0] mole_row_group,
402     input  [1:0] mole_col_group,
403     input      show,
404     output reg [7:0] row_out,
405     output reg [7:0] col_out
406 );
407
408     parameter FLIP_ROW = 0;
409     parameter FLIP_COL = 0;
410
411     wire [2:0] mole_row_raw = {mole_row_group, 1'b0};
412     wire [2:0] mole_col_raw = {mole_col_group, 1'b0};
413
414     wire [2:0] mole_row = FLIP_ROW ? (3'd7 - mole_row_raw) : mole_row_raw;
415     wire [2:0] mole_col = FLIP_COL ? (3'd7 - mole_col_raw) : mole_col_raw;
416
417     wire [2:0] real_row_sel = FLIP_ROW ? (3'd7 - row_sel) : row_sel;
418
419     always @(*) begin
420         row_out = 8'b11111111;
421         col_out = 8'b11111111;
422
423         row_out[real_row_sel] = 1'b0;
424
425         if (show && (real_row_sel == mole_row)) begin
426             col_out[mole_col] = 1'b0;
427         end
428     end
429
430 endmodule
431
432
433 // =====
434 // SEVEN SEG DECODER WITH LETTER SUPPORT
435 // =====
436 module SevenSeg(
437     input [3:0] val,
438     output reg [7:0] seg
439 );
440
441     always @(*) begin
442         case(val)

```

```

439 );
440
441 always @(*) begin
442     case(val)
443         4'd0: seg = 8'b11000000; // 0
444         4'd1: seg = 8'b11111001; // 1
445         4'd2: seg = 8'b10100100; // 2
446         4'd3: seg = 8'b10110000; // 3
447         4'd4: seg = 8'b10011001; // 4
448         4'd5: seg = 8'b10010010; // 5 S
449         4'd6: seg = 8'b10000010; // 6
450         4'd7: seg = 8'b11111000; // 7
451         4'd8: seg = 8'b10000000; // 8
452         4'd9: seg = 8'b10010000; // 9
453
454         4'hA: seg = 8'b10001000; // A or H style
455         4'hB: seg = 8'b11000111; // L
456         4'hC: seg = 8'b10000111; // T
457         4'hD: seg = 8'b10101111; // r-ish
458         4'hE: seg = 8'b10000110; // E
459         4'hF: seg = 8'b11111111; // blank
460
461         default: seg = 8'b11111111;
462     endcase
463 end
464
465 endmodule
466
467 // =====
468 // CLOCK DIVIDER 100 kHz
469 // =====
470
471 module ClockDivider100k(
472     input clk_in,
473     output reg clk_out
474 );
475
476     reg [8:0] count = 0;
477
478     always @(posedge clk_in) begin
479         if (count == 9'd249) begin
480             count <= 0;
481             clk_out <= ~clk_out;
482         end else begin
483             count <= count + 1;
484         end
485     end
486
487 endmodule
488
489 // =====
490 // Slow game clock: 100 kHz in to about 100 Hz out
491 // =====
492
493 module SlowGameClock(
494     input clk_in,
495     output reg clk_out
496 );

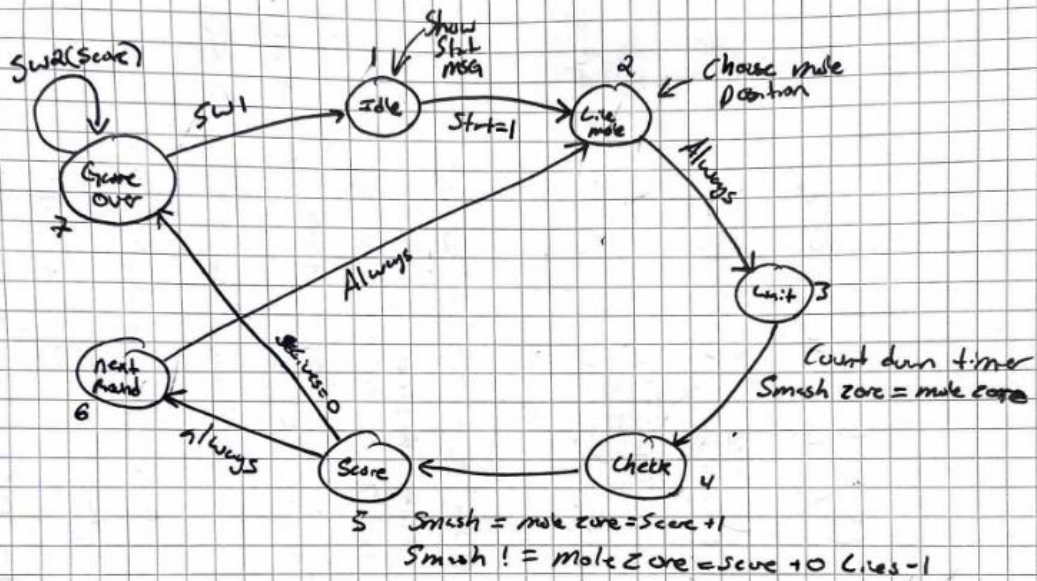
```

```

454         4'hA: seg = 8'b10001000; // A or H style
455         4'hB: seg = 8'b11000111; // L
456         4'hC: seg = 8'b10000111; // T
457         4'hD: seg = 8'b10101111; // r-ish
458         4'hE: seg = 8'b10000110; // E
459         4'hF: seg = 8'b11111111; // blank
460
461         default: seg = 8'b11111111;
462     endcase
463 end
464
465 endmodule
466
467
468 // =====
469 // CLOCK DIVIDER 100 kHz
470 // =====
471 module ClockDivider100k(
472     input clk_in,
473     output reg clk_out
474 );
475
476     reg [8:0] count = 0;
477
478     always @(posedge clk_in) begin
479         if (count == 9'd249) begin
480             count <= 0;
481             clk_out <= ~clk_out;
482         end else begin
483             count <= count + 1;
484         end
485     end
486
487 endmodule
488
489
490 // =====
491 // Slow game clock: 100 kHz in to about 100 Hz out
492 // =====
493 module SlowGameClock(
494     input clk_in,
495     output reg clk_out
496 );
497
498     reg [9:0] count = 0;
499
500     always @(posedge clk_in) begin
501         if (count == 10'd499) begin
502             count <= 0;
503             clk_out <= ~clk_out;
504         end else begin
505             count <= count + 1;
506         end
507     end
508
509 endmodule
510

```

State Machine Diagram:



States:

- 1: Idle
- 2: Light Mob
- 3: Wait
- 4: Check
- 5: Score
- 6: Next Round
- 7: Game Over

Buttons:

$$S_{W0} = S_{ht}$$
$$S_{w\Omega} = K_{\text{set}}$$

$S_{w2} = \text{Show Stts}$

Tricky Part of the project:

I think that the trickiest part of this project was firstly using the LED matrix as it was a little finicky and adapting it for use without a multiplexer, which I almost gave up on. I was able to get past this by carefully rewiring the board a few times using the pinout provided. Another issue I was having was deciding how the player would control the game at such a mole resolution of one LED, obviously 16 different buttons would've been cumbersome. So I think my quadrant approach is a good fit for ease of design and play. I was still able to use counters to control the timing to make the moles appear quicker but I decided against a win state as that would be having a set score limit and I thought it was more fun to have it as a "High Score" kind of game. So the only real display it will have is the start, lose and the high score via SW3. Overall this was a satisfying project once I got it working.