The Alligator Game

Robert Dermarkar - 998458057
Jing Wen Jiang - 1004145515
Station 28

Robert Dermarkar 998458057
Jing Wen Jiang 1004145515

Introduction

Our focus for the ECE241 final project was to implement an “Alligator Jaw Game” using the DE1-SoC Field Programmable Gate Array (FPGA). The Alligator Jaw Game is a test of the player’s reaction time, and begins when a player places their hand in the alligator’s jaws. At random, the alligator’s jaws will rapidly close on the player’s hand. If the player can escape in time, the game will continue and increment the player’s score. Conversely, when the player’s hand gets caught, the player loses, and the game is reset.

Though the game is conceptually simple, it required the implementation of a mechanical jaw system, as well as an accompanying VGA display. The visual and mechanical peripheral components of the project were paired with the game’s central logic, and controlled through the DE1-SoC FPGA. The implementation of the game and its peripheral systems demonstrated the versatility of the FPGA in its applications, and provided us with an opportunity to work on a project that involved both coding and mechanical work.

The Design

Top Level Overview

Prior to the start of the game, the VGA displays an image of an open alligator’s jaw (Figure 1.1). The game begins when the user places their hand in the jaw of the alligator, and presses the start button located at its throat. Once pressed, the VGA changes its display (Figure 1.2), and a random countdown begins. When the countdown is complete, the central game controller signals the motors controlling the alligator’s jaws to close.

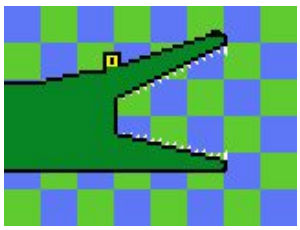


Figure 1.1

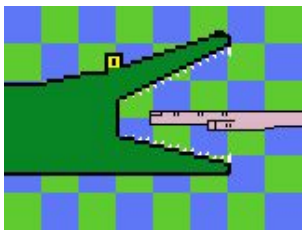


Figure 1.2

Figure 1.3

Figure 1.4

Figure 1.5

When the jaw closes, two outcomes are possible: if the player is able to escape the alligator's jaws before it closes, the player wins the round and an incremented score is displayed on the top right corner of the VGA (Figure 1.3). The player will then have 15 seconds to re-press the start button to continue the game. Else, the game and score will reset. If the user's hand is caught, the player loses, and the game also resets (Figure 1.4-1.5).

In order to detect a win or loss, a limit switch is mounted to detect when the alligator's jaw is closed (Figure 2). If the player escaped, the closing limit switch will be depressed by the jaw. However, if the jaw closes on the player's hand, the closing limit switch remains unpressed. In the event that the player pulls out their hand after being caught by the alligator's jaws, a quarter-second counter is implemented: the closing limit switch must be pressed within this time limit. If not, it indicates that the jaw's closing movement was hindered by the player, forcing its normal closing time to be delayed.

Immediately following win-loss detection, the motors controlling the alligator's jaw will receive an opening signal from the central game control. The motors will continue to receive the signal until the opening limit switch is engaged, indicating that the jaws have been fully opened.

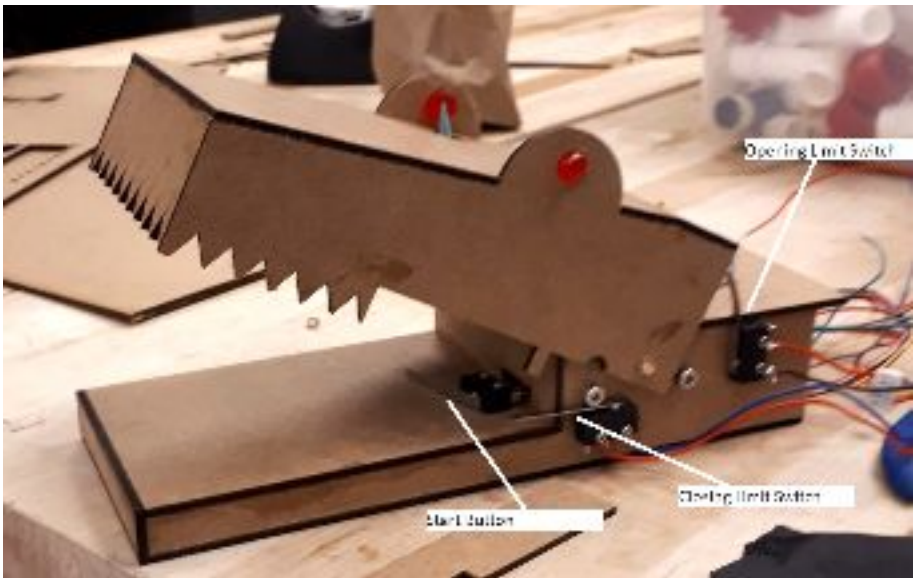


Figure 2. Limit switches on alligator jaw.

Central Game Control

The central game control is a finite state machine (FSM) that detects the state of the game depending on inputs from the limit switches, and from the time elapsed. The controller signals to the motor FSM and VGA FSM to produce visual and mechanical outputs appropriate for each stage of the game.

There are twelve distinct states in the central game FSM (for detailed state transitions and descriptions refer to Appendix A). The game cycles in the PRESTART state until the start button is
pressed. Once the start button is pressed the FSM moves on the a countdown with a random duration under 10 seconds at the end of which it closes. It then goes to the CHECK state where it looks to so if the limit switch is closed, if so the WIN state is initiated. After the WIN state the jaw will open and the FSM goes to the READY state waiting for the player to play again. If the limit switch was not closed during the CHECK state the FSM goes to the LOSE state and the game reset going back to the PRESTART state with the jaw open.

Graphics Control

The graphics control is responsible for choosing which images will be displayed. The FSM is comprised of 13 states, with a DRAW and WAIT state for each image as well as a HOLD state for idling. The FSMs decides which image to draw based on its current current as well as the current state of the central game control FSM. The state of the graphics control FSM is then used the determine which memory cells in the ROM should be accessed to draw the games animations.

Motor Control

The motor receives signals from the central FSM to open or close the jaw. To close it, the motor FSM outputs high and low signals in a particular order to induce the coils within the unipolar stepper motors. To open, the motor FSM receives a cue from the central control, and outputs to the pins on the FPGA. The opening sequence induces the stepper motor coils in a pattern opposite to that of close.

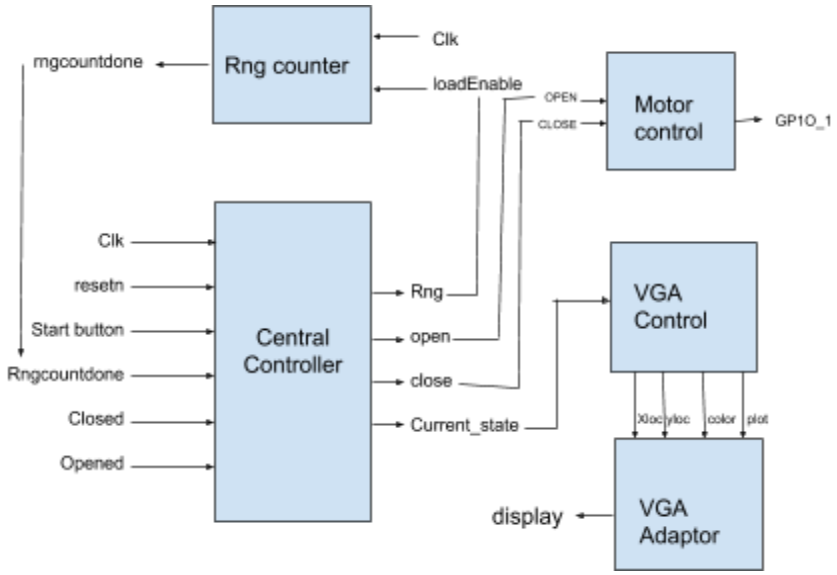


Figure 3. Simplified block diagram. For detailed diagram, reference Appendix B

Robert Dermarkar 998458057
Jing Wen Jiang 1004145515

Report on Success

Several issues were present when implementing the peripheral systems of The Alligator Game. When working on pairing the VGA display with the progress of the game, the first problem encountered was correctly outputting the desired pattern onto the screen.

In order to allow for an incrementing score on the display, numbers had to be drawn individually. To achieve this, the VGA display was broken into 2 distinct parts: a strip at the top for the scoreboard, and a large section below it for the game's the main images. Drawing the numbers in the top strip required that the strip be subdivided into squares, each of which would be size-appropriate to fit the numbers 0-9.

Due to our decision to segment the screen, it was important that the VGA was drawing correct patterns onto correct segments of the screen. To help keep track of locations on the screen, a checkered background was chosen for the game graphics (Figure 1).

Obtaining correct output from the VGA was further complicated by the inability to access correct memory blocks: instead of accessing the first cell of the memory block, memory from three before the desired cells were always accessed. This issue made the issue of correctly outputting patterns on segments of the display difficult to diagnose and debug.

We were able to overcome these issues through trial and error. As graphics information for each pixel is stored in the form of a 24-bit stream of one's and zero's, it is difficult to recognize the problem without seeing the graphic output. Thus, the most effective debugging method was through observation, rather than carefully simulating the circuit output.

The final and most difficult issue to overcome in the VGA FSM was our inability to predict its behaviour. Though the state transitions for the FSM controlling the VGA was clearly defined in the state table (Appendix), the FSM transitioned into unexpected states when minor changes in the code were made. In order to ensure the correspondence of VGA output to the current game state, a hex display was used in debugging to help correlate the main FSM's states with the VGA FSM's states.

We managed to have VGA output that corresponded to when the game is waiting for a player, when the player presses the start button, and "ESCAPED!" and "YOU LOSE!" screens to indicate to the player if they have won or lost (Figure 1). However, we were unable to achieve our original plans to flash Figure 1.1 and Figure 1.2 across the screen when waiting for a player to press the start button.

On the mechanical side, we experienced troubles with opening and closing the alligator's jaws. Although simulations showed that the motors were correctly controlled, two motors working in conjunction had great troubles closing the jaw, and were completely unable to open the jaw. By isolating and testing each motor individually, it was found that one motor driver had become

defective. Moreover, through trial and error, the speed of opening and closing the jaw was optimized through manipulating the speed of coil induction and the number of coils induced in the stepper motors.

Overall, the challenges presented by the project were mostly resolved. The mechanical jaw can close rapidly, and can open without human intervention. The VGA display, although missing flashing start-up graphics, was able to reflect the progress of the game effectively, and the missing graphics did not take away from the overall experience of the game.

What to do Differently

When designing the mechanical jaw system, we opted to utilize two unipolar stepper motors to move the jaws. Though unipolar stepper motors are easier to operate, they also provided less torque than bipolar stepper motors. This became a particular issue as the mechanical system was subject to frequent testing. We noticed that as time passed, the stepper motors became weaker, and jaw movement slowed. On the day of the demonstration, the motors were unable to turn and move the jaw effectively, forcing us to make edits to the code in order to resume the previous functionality of the jaws.

Though unipolar stepper motors were easier to operate initially, their performance was unreliable over time. Given the chance to rebuild the mechanical component, we would choose to use bipolar stepper motors for their increased torque and reliability.

Additionally, the limit switches utilized in the mechanical system were unreliable, and key presses were were not consistently being registered. As a result, the start button sometimes required being pressed multiple times before being detected.

In the current mechanical system, switches are bolted and soldered onto the jaw. This design choice makes it difficult to replace defective parts. If we were to rebuild this design, all sensors and electronic parts would be attached to the system in a way that allows for the ease of replacement.

We designed the circuit with the philosophy of writing each part to function, and only then modifying it to operate with other modules. This was a mistake as if required constantly going back and altering parts of the design which were completed. These changes opened the potential for breaking functional circuits. Another issue with this approach was it made it harder to predict how the FSMs' behaviours would interact, and may have led to several of our issues with the VGA FSM as it was the last one written. If we were to rebuild the logic circuit, we would begin with the end goal in mind: we would plan how the three FSMs would work as a unit, rather than writing independently functioning modules that needed additional modifications in order to work together.

Robert Dermarkar 998458057
Jing Wen Jiang 1004145515

Appendix

Appendix A - Verilog code used to implement The Alligator Game

```
1   //Main Game Control
2
3   module maingamecontrol
4         (
5            CLOCK_50,                        // On Board 50 MHz
6            KEY,                             // On Board Keys
7            GPIO_0,
8            GPIO_1,
9            VGA_CLK,                         // VGA Clock
10           VGA_HS,                          // VGA H_SYNC
11           VGA_VS,                          // VGA V_SYNC
12           VGA_BLANK_N,                        // VGA BLANK
13           VGA_SYNC_N,                      // VGA SYNC
14           VGA_R,                           // VGA Red[9:0]
15           VGA_G,                           // VGA Green[9:0]
16           VGA_B                         // VGA Blue[9:0]
17        );
18
19        input            CLOCK_50;              // 50 MHz
20        input  [3:0]     KEY;
21        input  [5:0]     GPIO_0;
22        output [20:0]    GPIO_1;
23        output           VGA_CLK;               // VGA Clock
24        output           VGA_HS;                // VGA
    H_SYNC
25        output           VGA_VS;                // VGA
    V_SYNC
26        output           VGA_BLANK_N;           // VGA BLANK
27        output           VGA_SYNC_N;            // VGA SYNC
28        output [7:0]     VGA_R;                 // VGA
    Red[7:0] Changed from 10 to 8-bit DAC
29        output [7:0]     VGA_G;                 // VGA
    Green[7:0]
30        output [7:0]     VGA_B;                 // VGA
    Blue[7:0]
31
32
33
34        //declaring wires for external inputs
35        wire startbutton;
36        wire resetn;
37        wire closed;
38        wire opened;
39
40        //assigning external inputs
41        assign resetn = KEY[0];
42        assign startbutton = ~GPIO_0[0];
```

```
43          assign closed = ~GPIO_0[1];
44          assign opened = ~GPIO_0[2];
45
46          //declaring wires for external outputs
47          wire [3:0] motorpins;
48
49          //assigning external outputs
50          assign GPIO_1[3:0] = motorpins[3:0];
51
52          //declaring wires for FSM
53          wire close, open, rng, prestart, win, ready;
54          wire rngcountdone;
55          wire [3:0] current_state;
56          wire hand_out;
57
58          //declaring wires for countdown controls
59          wire countdone_quarter, countdone_half,
    countdone_one, countdone_seven, countdone_two,
    countdone_four;
60          wire enable_quarter, enable_half, enable_seven,
     enable_one, enable_two, enable_four;
61
62          //declaring wires for ramdon number look-up list
63          wire [3:0] rngnum;
64          wire [32:0] rngnumout;
65          wire [4:0] score1, score2, score3;
66
67          //declaring wires for motorcontrol
68          wire jaw_is_open, jaw_is_closed;
69          wire [3:0] current;
70          wire draw;
71
72          //declaring wires for VGA control
73          wire [7:0] x;
74          wire [6:0] y;
75          wire [23:0] colour;
76          wire [3:0] currentVGA;
77
78          //Declaring Video output module
79          vga_adapter VGA(.resetn(resetn),
80                          .clock(CLOCK_50),
81                          .colour(colour),
82                          .x(x),
83                          .y(y),
84                          .plot(1),
85                          /* Signals for the DAC to
    drive the monitor. */
```

```
 86                              .VGA_R(VGA_R),
 87                              .VGA_G(VGA_G),
 88                              .VGA_B(VGA_B),
 89                              .VGA_HS(VGA_HS),
 90                              .VGA_VS(VGA_VS),
 91                              .VGA_BLANK(VGA_BLANK_N),
 92                              .VGA_SYNC(VGA_SYNC_N),
 93                              .VGA_CLK(VGA_CLK));
 94          defparam VGA.RESOLUTION = "160x120";
 95          defparam VGA.MONOCHROME = "FALSE";
 96          defparam VGA.BITS_PER_COLOUR_CHANNEL  = 8;
 97          defparam VGA.BACKGROUND_IMAGE =
     "StartingBackground.mif" ;
 98
 99
100          //declaring motorFSM module
101          motorFSM m0(.clock(CLOCK_50),
102                      .resetn(resetn),
103                      .open_fsm(open),
104                      .close_fsm(close),
105                      .limit_jawOpen(opened),
106                      .limit_jawClose(closed),
107                      .jaw_is_open(jaw_is_open),
108                      .jaw_is_closed(jaw_is_closed),
109                      .pin(motorpins),
110                      .current(current)
111                     );
112
113          //declaring main game FSM module
114          controller c0(.clk(CLOCK_50),
115                        .resetn(resetn),
116                        .startbutton(startbutton),
117                        .rngcountdone(rngcountdone),
118                        .hand_out(hand_out),
119                        .closed(closed),
120                        .prestart(prestart),
121                        .close(close),
122                        .open(open),
123                        .rng(rng),
124                        .win(win),
125                        .ready(ready),
126                        .current_state(current_state),
127                        .countdone_quarter(
     countdone_quarter),
128                        .countdone_half(countdone_half),
129                        .countdone_one(countdone_one),
130                        .countdone_two(countdone_two),
```

```
131                         .countdone_four(countdone_four),
132                         .countdone_seven(countdone_seven),
133                         .enable_quarter(enable_quarter),
134                         .enable_half(enable_half),
135                         .enable_one(enable_one),
136                         .enable_two(enable_two),
137                         .enable_four(enable_four),
138                         .enable_seven(enable_seven),
139                         .opened(opened)
140                         );
141
142         //declaring main game control datapath
143         datapath d0(.clk(CLOCK_50),
144                     .prestart(prestart),
145                     .win(win),
146                     .closed(closed),
147                     .close(close),
148                     .check(check),
149                     .score1(score1),
150                     .score2(score2),
151                     .score3(score3),
152                     .hand_out(hand_out),
153                     .ready(ready)
154                     );
155
156
157         //declaring Video output FSM module
158         VGAcontrol VGA0(.clk(CLOCK_50),
159                     .resetn(resetn),
160                     .start(rng),
161                     .score1(score1),
162                     .score2(score2),
163                     .score3(score3),
164                     .x_out(x),
165                     .y_out(y),
166                     .colour_out(colour),
167                     .draw(draw),
168                     .currentmain(current_state)
169                     );
170
171         //declaring random number cycling module
172         rng r0(.clk(CLOCK_50),
173                .reset_n(resetn),
174                .rngnum(rngnum)
175               );
176
177         //declaring random number lookup table module
```

```
178         randomnumberlookup  r1(.rngnum(rngnum),
179                                .rngnumout(rngnumout)
180                                );
181
182         //declaring countdown from ramdom number module
183         rngcountdown  c1(.clk(CLOCK_50),
184                         .loadEnable(rng),
185                         .load(rngnumout),
186                         .countDone(rngcountdone)
187                         );
188
189         //declaring a variety of countdown modules
190         countdown_half  c2(.clk(CLOCK_50),
191                         .loadEnable(enable_half),
192                         .countDone(countdone_half)
193                         );
194
195         countdown_quarter  c3(.clk(CLOCK_50),
196                               .loadEnable(enable_quarter),
197                               .countDone(
    countdone_quarter)
198                               );
199
200         countdown_seven  c4(.clk(CLOCK_50),
201                             .loadEnable(enable_seven),
202                             .countDone(countdone_seven)
203                             );
204
205         countdown_four  c6(.clk(CLOCK_50),
206                             .loadEnable(enable_four),
207                             .countDone(countdone_four)
208                             );
209
210         countdown_two  c7(.clk(CLOCK_50),
211                             .loadEnable(enable_two),
212                             .countDone(countdone_two)
213                             );
214
215         countdown_one  c8(.clk(CLOCK_50),
216                             .loadEnable(enable_one),
217                             .countDone(countdone_one)
218                             );
219
220     endmodule
221
222
223     //This module is an FSM which acts as the brain of
```

```
    the game
224 //it determines when the game starts, when the jaw
    closes
225 //and if the player won or lost
226 module controller(
227                 input clk,
228                 input resetn,
229                 input startbutton,
230                 input rngcountdone, hand_out,
231                 input closed, opened,
232                 input countdone_quarter,
    countdone_half, countdone_one, countdone_two,
    countdone_four, countdone_seven,
233                 output reg prestart, close, open, rng,
     win, ready, gameover,
234                 output reg enable_quarter, enable_half
    , enable_one, enable_two, enable_four, enable_seven,
235                 output reg [3:0] current_state
236                 );
237
238    reg [3:0] next_state;
239
240    //declares states
241    localparam  PRESTART    = 4'd0,
242                WAIT        = 4'd1,
243                CLOSE       = 4'd2,
244                CHECK       = 4'd3,
245                WIN         = 4'd4,
246                OPEN        = 4'd5,
247                AFTERWIN    = 4'd6,
248                READY       = 4'd7,
249                LOSE        = 4'd8,
250                RESETGAME   = 4'd9,
251                AFTERLOSE   = 4'd10,
252                GAMEOVER    = 4'd11;
253
254     //this section dictates the order of states and
    the
255     //requirements for switching between states
256     always@(*)
257     begin: state_table
258     case(current_state)
259          PRESTART: begin
260             if(startbutton) next_state = WAIT;
    //checks if startbutton pressed if so goes to WAIT
    state
261          end
```

```
262             WAIT: begin
263                 if(rngcountdone)  next_state = CLOSE;
    //checks if the random countdown is done if so jaw
    closes
264             end
265             CLOSE: begin
266                 if(countdone_one) next_state = CHECK;
    //waits one second as the jaw closes then checks if
    the jaw has closed
267             end
268             CHECK: begin
269                 if(countdone_quarter) begin   //checks
    if jaws caught hand
270                     if(closed || hand_out > 0 ) next_state
     = WIN;       //closed represents the limit switch if
    the limit switch is closed the player wins
271                     else if(!closed) next_state = LOSE;
        //If the limit switch is open the player loses
272                 end
273             end
274             LOSE: begin
275                 next_state = RESETGAME;  //continues
    directly to resetgame
276             end
277             AFTERLOSE: begin
278                 if(countdone_two) next_state = GAMEOVER;
      //waits 2 seconds then contines to GAMEOVER this
    is to allow for the You Lose screen to show
279             end
280             GAMEOVER: begin
281                 if(countdone_four) next_state = PRESTART;
     //waits 4 second while showing the GAME OVER screen
    the goes back to the PRESTART state
282             end
283             RESETGAME: begin
284                 if(opened || countdone_four)  //waits
    for the jaw to hit the open limit switch or for 4
    seconds to protect the motor
285                 next_state = AFTERLOSE;  //moves to
    afterlose state
286             end
287             WIN: begin
288                 next_state = OPEN;  //continues directly
    to open state
289             end
290             OPEN: begin
291                 if(opened || countdone_one)   next_state
```

```
= AFTERWIN;  //Waits For Jaws to open or for 1 second
292             end
293             AFTERWIN: begin
294                 if(countdone_two) next_state = READY;
    //waits 2 seconds while Escaped screen shown
295             end
296             READY: begin
297                 if(startbutton) next_state = WAIT;
    //checks if startbutton pressed if so goes to WAIT
    state
298                 else if(countdone_seven) next_state =
    PRESTART;  //Waits 7 seconds then goes to PRESTART
299             end
300             default: next_state = PRESTART;
301         endcase
302         end
303
304         //This section dictates what will happen in each
    state
305         //as well as starting countdowns
306         always @(*)
307         begin: enable_signals
308
309           prestart        <= 0;
310           rng             <= 0;
311           close           <= 0;
312           open            <= 0;
313           win             <= 0;
314           ready           <= 0;
315           enable_half     <= 0;
316           enable_quarter  <= 0;
317           enable_one      <= 0;
318           enable_two      <= 0;
319           enable_four     <= 0;
320           enable_seven    <= 0;
321
322             case (current_state)
323                 PRESTART: begin   //the prestart state
    ensures that all game values are in their starting
    state
324                     prestart        <= 1;
325                     rng             <= 0;
326                     close           <= 0;
327                     open            <= 0;
328                     win             <= 0;
329                     ready           <= 0;
330                     enable_half     <= 0;
```

```
331                     enable_quarter  <= 0;
332                     enable_two      <= 0;
333                     enable_four     <= 0;
334                     enable_seven    <= 0;
335                 end
336                 WAIT: begin         //wait begins the
    ramdon countdown
337                     prestart        <= 0;
338                     rng             <= 1;
339                     close           <= 0;
340                     open            <= 0;
341                     win             <= 0;
342                     ready           <= 0;
343                     enable_half     <= 0;
344                     enable_quarter  <= 0;
345                     enable_two      <= 0;
346                     enable_four     <= 0;
347                     enable_seven    <= 0;
348                 end
349                 CLOSE: begin        //close closes the jaw
350                     prestart        <= 0;
351                     rng             <= 0;
352                     close           <= 1;
353                     open            <= 0;
354                     win             <= 0;
355                     ready           <= 0;
356                     enable_half     <= 0;
357                     enable_quarter  <= 0;
358                     enable_one      <= 1;
359                     enable_two      <= 0;
360                     enable_four     <= 0;
361                     enable_seven    <= 0;
362                 end
363                 OPEN: begin         //open opens the jaw
364                     prestart        <= 0;
365                     rng             <= 0;
366                     close           <= 0;
367                     open            <= 1;
368                     win             <= 0;
369                     ready           <= 0;
370                     enable_half     <= 0;
371                     enable_quarter  <= 0;
372                     enable_one      <= 1;
373                     enable_two      <= 0;
374                     enable_four     <= 0;
375                     enable_seven    <= 0;
376                 end
```

```
377                 CHECK: begin        //check starts a
    counter after which the FSM checks if the player won
    or lost
378                     prestart        <= 0;
379                     rng             <= 0;
380                     close           <= 0;
381                     open            <= 0;
382                     win             <= 0;
383                     ready           <= 0;
384                     enable_half     <= 0;
385                     enable_quarter  <= 1;
386                     enable_one      <= 0;
387                     enable_two      <= 0;
388                     enable_four     <= 0;
389                     enable_seven    <= 0;
390                 end
391                 WIN: begin          //sends the win signal
    to the datapath incrementing the score
392                     prestart        <= 0;
393                     rng             <= 0;
394                     close           <= 0;
395                     open            <= 0;
396                     win             <= 1;
397                     ready           <= 0;
398                     enable_half     <= 0;
399                     enable_quarter  <= 0;
400                     enable_two      <= 0;
401                     enable_four     <= 0;
402                     enable_seven    <= 0;
403                 end
404                 AFTERWIN: begin     //starts 2 second
    counter for video purposes
405                     prestart        <= 0;
406                     rng             <= 0;
407                     close           <= 0;
408                     open            <= 0;
409                     win             <= 0;
410                     ready           <= 0;
411                     enable_half     <= 0;
412                     enable_quarter  <= 0;
413                     enable_two      <= 1;
414                     enable_four     <= 0;
415                     enable_seven    <= 0;
416                 end
417                 LOSE: begin
418                     prestart        <= 0;
419                     rng             <= 0;
```

```
420                     close           <= 0;
421                     open            <= 0;
422                     win             <= 0;
423                     ready           <= 0;
424                     enable_half     <= 0;
425                     enable_quarter  <= 0;
426                     enable_two      <= 0;
427                     enable_four     <= 0;
428                     enable_seven    <= 0;
429                 end
430                 AFTERLOSE: begin  //starts the 2 second
    counter for video purposes
431                     prestart        <= 0;
432                     rng             <= 0;
433                     close           <= 0;
434                     open            <= 0;
435                     win             <= 0;
436                     ready           <= 0;
437                     enable_half     <= 0;
438                     enable_quarter  <= 0;
439                     enable_two      <= 1;
440                     enable_four     <= 0;
441                     enable_seven    <= 0;
442                 end
443                 GAMEOVER: begin   //starts the 4 second
    timer for video purposes
444                     prestart        <= 0;
445                     rng             <= 0;
446                     close           <= 0;
447                     open            <= 0;
448                     win             <= 0;
449                     ready           <= 0;
450                     enable_half     <= 0;
451                     enable_quarter  <= 0;
452                     enable_two      <= 0;
453                     enable_four     <= 1;
454                     enable_seven    <= 0;
455                 end
456                 RESETGAME: begin  //opens the jaw and
    satrts 4 second timer
457                     prestart        <= 0;
458                     rng             <= 0;
459                     close           <= 0;
460                     open            <= 1;
461                     win             <= 0;
462                     ready           <= 0;
463                     enable_half     <= 0;
```

```
464                     enable_quarter  <= 0;
465                     enable_two      <= 0;
466                     enable_four     <= 1;
467                     enable_seven    <= 0;
468                 end
469                 READY: begin      //sends ready signal
    to datapath and starts 7 second timer
470                     prestart        <= 0;
471                     rng             <= 0;
472                     close           <= 0;
473                     open            <= 0;
474                     win             <= 0;
475                     ready           <= 1;
476                     enable_half     <= 0;
477                     enable_quarter  <= 0;
478                     enable_two      <= 0;
479                     enable_four     <= 0;
480                     enable_seven    <= 1;
481                 end
482             endcase
483         end // enable_signals
484
485         //sets state transition to clock edge
486         always@(posedge clk)
487         begin: state_FFs
488             if(!resetn)
489                 current_state  = PRESTART;
490             else
491                 current_state  = next_state;
492         end // state_FFS
493     endmodule
494
495     //the datapath 2 concerned with 2 functions
496     //1.incrementing the score
497     //2.ensuring that if the jaw fully closes then opens
    slightly the player will still win
498     module datapath(  clk,
499                       prestart,
500                       win,
501                       closed,
502                       close,
503                       check,
504                       ready,
505                       score1,
506                       score2,
507                       score3,
508                       hand_out
```

```
509                         );
510
511         input clk;
512         input prestart;
513         input win;
514         input closed;
515         input close;
516         input check;
517         input ready;
518         output reg [4:0] score1, score2, score3;
519         output reg hand_out;
520
521         //score incrementer
522         always@(posedge clk)
523         begin
524             //sets score to 0 if the game is in prestart
        state
525             if(prestart) begin
526                 score1 <= 4'b0;
527                 score2 <= 4'b0;
528                 score3 <= 4'b0;
529             end
530             //if the win value is true the score is
        incremented by one
531             //to make graphics display easier the score
        value is broken
532             //into 3 parts and each is kept between 0-9
533             else if(win) begin
534                 if(score1 < 9) score1 <= score1 + 1;
535                 else if(score1 == 9) begin
536                     score1 <= 0;
537                     if(score2 < 9) score2 <= score2 + 1;
538                     else if(score2 == 9) begin
539                         score2 <= 0;
540                         score3 <= score3 + 1;
541                     end
542                 end
543             end
544         end
545
546         //ensures that if the jaw fully closes then opens
        slightly the player will still win
547         always@(*)
548         begin
549             //sets handout to 0 after check states is over
        effectively
550             if(prestart || ready)
```

```
551             hand_out <= 0;
552         //sets hand_out to 1 if jaw closes during
    close or check state
553         else if(close || check) begin
554             if(closed && hand_out == 0)
555                 hand_out <= 1;
556         end
557     end
558 endmodule
559
560
561 //selects a number based on a 4 bit input
562 //in this circuit the 4 bit value in being fed in by
    a constantly cycling counter
563 module randomnumberlookup( input [3:0]rngnum, output
    reg[32:0]rngnumout);
564
565     always@(*)
566     begin
567     case(rngnum[3:0])
568         4'd0:    rngnumout = 'd50000000;
569         4'd1:    rngnumout = 'd106250000;
570         4'd2:    rngnumout = 'd134375000;
571         4'd3:    rngnumout = 'd162500000;
572         4'd4:    rngnumout = 'd190625000;
573         4'd5:    rngnumout = 'd218750000;
574         4'd6:    rngnumout = 'd246875000;
575         4'd7:    rngnumout = 'd275000000;
576         4'd8:    rngnumout = 'd303125000;
577         4'd9:    rngnumout = 'd331250000;
578         4'd10:   rngnumout = 'd359375000;
579         4'd11:   rngnumout = 'd387500000;
580         4'd12:   rngnumout = 'd415625000;
581         4'd13:   rngnumout = 'd443750000;
582         4'd14:   rngnumout = 'd471875000;
583         4'd15:   rngnumout = 'd500000000;
584         default: rngnumout = 'd50000000;
585     endcase
586     end
587 endmodule
588
589 //cycles 4-bit value at clock edge
590 module rng(input clk, reset_n, output reg [3:0]rngnum
    );
591      always @ (posedge clk) begin
592          rngnum <= rngnum + 1;
593      end
```

```
594    endmodule
595
596    //counts down from loaded in random value
597    module rngcountdown(clk, load, loadEnable, countDone);
598       input clk, loadEnable;
599       input [32:0]load;
600       output reg countDone;
601
602       reg [32:0]countVal;
603
604       always @(posedge clk) begin
605          //if the loadEnable value is false random
       value is loaded in
606          //(a little confusing i know)
607          //and countDone set to zero
608          if (!loadEnable) begin
609             countVal <= load;
610             countDone <= 0;
611          end
612
613          //when the value is counted down the zero
       countDone is set to 1
614          else if (countVal == 'd0) begin
615             countDone <= 1;
616          end
617
618          //while the value is not zero it is
       incremented down each clock cycle
619          else if(countVal != 'd0) begin
620             countVal <= countVal - 1;
621             countDone <= 0;
622          end
623       end
624    endmodule
625
626    ////////////////////////////////////////////////////////
       ////////////////////////////////
627    ////////////Collection of counters below only first
       one will be commented////////////
628    ////////////////////////////////////////////////////////
       ////////////////////////////////
629
630    module countdown_one(clk, loadEnable, countDone);
631       input clk, loadEnable;
632       output reg countDone;
633
634       reg [32:0]countVal;
```

```
635
636         always @(posedge clk) begin
637         //while loadEnable = 0 countval is set to initail
        value and countDone is set to 0
638         if (!loadEnable) begin
639             countVal <= 'd50000000;
640             countDone <= 0;
641         end
642
643         //when countval is equal to 0 countDone is set to 1
644         else if (countVal == 'd0) begin
645             countDone <= 1;
646         end
647
648         //while countval is not zero it is incremented
        down each clock cycle
649         else if(countVal != 'd0) begin
650             countVal <= countVal - 1;
651             countDone <= 0;
652         end
653         end
654     endmodule
655
656
657     module countdown_half(clk, loadEnable, countDone);
658          input clk, loadEnable;
659          output reg countDone;
660
661          reg [32:0]countVal;
662
663          always @(posedge clk) begin
664              if (!loadEnable) begin
665             countVal <= 'd25000000;
666             countDone <= 0;
667         end
668
669         else if (countVal == 'd0) begin
670             countDone <= 1;
671         end
672
673         else if(countVal != 'd0) begin
674             countVal <= countVal - 1;
675             countDone <= 0;
676         end
677          end
678     endmodule
679
```

```
680   module countdown_quarter(clk, loadEnable, countDone);
681       input clk, loadEnable;
682       output reg countDone;
683
684       reg [32:0]countVal;
685
686      always @(posedge clk) begin
687           if (!loadEnable) begin
688         countVal <= 'd12500000;
689         countDone <= 0;
690      end
691
692           else if (countVal == 'd0) begin
693         countDone <= 1;
694      end
695
696      else if(countVal != 'd0) begin
697         countVal <= countVal - 1;
698         countDone <= 0;
699      end
700      end
701   endmodule
702
703   module countdown_two(clk, loadEnable, countDone);
704       input clk, loadEnable;
705       output reg countDone;
706
707       reg [32:0]countVal;
708
709      always @(posedge clk) begin
710           if (!loadEnable) begin
711         countVal <= 'd100000000;
712         countDone <= 0;
713      end
714
715      else if (countVal == 'd0) begin
716            countDone <= 1;
717      end
718
719      else if(countVal != 'd0) begin
720         countVal <= countVal - 1;
721         countDone <= 0;
722      end
723      end
724   endmodule
725
726
```

```
727   module countdown_four(clk, loadEnable, countDone);
728       input clk, loadEnable;
729       output reg countDone;
730   
731       reg [32:0]countVal;
732   
733      always @(posedge clk) begin
734           if (!loadEnable) begin
735         countVal <= 'd200000000;
736         countDone <= 0;
737      end
738   
739      else if (countVal == 'd0) begin
740            countDone <= 1;
741      end
742   
743      else if(countVal != 'd0) begin
744         countVal <= countVal - 1;
745         countDone <= 0;
746      end
747      end
748   endmodule
749   
750   
751   module countdown_seven(clk, loadEnable, countDone);
752       input clk, loadEnable;
753       output reg countDone;
754   
755       reg [32:0]countVal;
756   
757      always @(posedge clk) begin
758           if (!loadEnable) begin
759         countVal <= 'd750000000;
760         countDone <= 0;
761      end
762   
763      else if (countVal == 'd0) begin
764            countDone <= 1;
765      end
766   
767      else if(countVal != 'd0) begin
768         countVal <= countVal - 1;
769         countDone <= 0;
770      end
771      end
772   endmodule
773   
```

```
1   //VGA control module
2
3   module VGAcontrol(clk,
4                     resetn,
5                     start,
6                     score1,
7                     score2,
8                     score3,
9                     x_out,
10                    y_out,
11                    colour_out,
12                    draw,
13                    currentmain
14                    );
15
16      input resetn;
17      input clk;
18      input start;
19      input [3:0] score1, score2, score3;
20      input [3:0] currentmain;
21      output [7:0] x_out;
22      output [6:0] y_out;
23      output [23:0] colour_out;
24      output draw;
25
26      //declares wires for drawcontrol
27      wire countdone;
28      wire draw_done;
29      wire countstart;
30      wire [3:0] current;
31      wire pre;
32
33      //declares wires for drawdatapath
34      wire [23:0] imagedata;
35      wire [17:0] address_start;
36      wire [11:0] num1_start, num2_start, num3_start;
37      wire [23:0] numberdata1, numberdata2, numberdata3;
38      wire [7:0] x_counter;
39      wire [6:0] y_counter;
40      wire [7:0] x_out;
41      wire [6:0] y_out;
42      wire [17:0] address;
43
44
45      wire [11:0] num1address, num2address, num3address;
46      wire [8:0] pixel;
47
```

```
48      //pixel describes the position of the pixel being drawn in 20x20 blocks at the top of the screen
49      //this is used with num#_start to determine which part of the rom should be accessed
50      //used to draw the score numbers
51      assign pixel = (y_counter * 20) + x_counter + 2;
52
53      //look_up table module to find address start value using score values
54      numberSelect l1(.num_select(score1), .address_num(num1_start));
55      numberSelect l2(.num_select(score2), .address_num(num2_start));
56      numberSelect l3(.num_select(score3), .address_num(num3_start));
57
58      //assigns the address to access the rom using the pixel value and start value from lookup table
59      assign num1address = num1_start + pixel;
60      assign num2address = num2_start + pixel;
61      assign num3address = num3_start + pixel;
62
63      //pulls colour data from rom based on num#address
64      ROMnumbers ROM1(.address(num1address), .clock(clk), .q(numberdata1));
65      ROMnumbers ROM2(.address(num2address), .clock(clk), .q(numberdata2));
66      ROMnumbers ROM3(.address(num3address), .clock(clk), .q(numberdata3));
67
68      wire [17:0] imageaddress;
69
70      //gets rom address start position based on FSM output
71      backSelect back1(.back_select(current), .address_back(address_start));
72
73      //calculates current draw position based on y_out and x_out values
74      assign imageaddress = address_start + ((y_out - 20) * 160) + x_out + 3;
75
76      //pulls colour data from rom based on imageaddress
77      ROMbackground ROM0(.address(imageaddress), .clock(clk), .q(imagedata));
78
79      //declares fsm for the video output
```

```
 80      drawcontrol dc(.clk(clk),
 81                     .resetn(resetn),
 82                     .start(start),
 83                     .pre(pre),
 84                     .countdone(countdone),
 85                     .draw_done(draw_done),
 86                     .draw(draw),
 87                     .countstart(countstart),
 88                     .current(current),
 89                     .currentmain(currentmain));
 90
 91      //declares datapath for the video output
 92      drawdatapath(  .clk(clk),
 93                     .resetn(resetn),
 94                     .pre(pre),
 95                     .draw(draw),
 96                     .imagedata(imagedata),
 97                     .numberdata1(numberdata1),
 98                     .numberdata2(numberdata2),
 99                     .numberdata3(numberdata3),
100                     .colour_out(colour_out),
101                     .x_out(x_out),
102                     .y_out(y_out),
103                     .address(address),
104                     .draw_done(draw_done),
105                     .x_counter(x_counter),
106                     .y_counter(y_counter)
107                     );
108
109      //declares 2 second countdown module
110      countdown_2 cd3(.clk(clk), .loadEnable(countstart
     ), .countDone(countdone));
111
112  endmodule
113
114
115  //the drawcontrol module is the FSM for the Video
     Output
116  //it uses the state of the main FSM to determine
     which image should be shown
117  module drawcontrol(clk,
118                     resetn,
119                     start,
120                     pre,
121                     countdone,
122                     draw_done,
123                     draw,
```

```
124                     countstart,
125                     current,
126                     currentmain);
127
128     input clk;
129     input resetn;
130     input start;
131     input countdone;
132     input draw_done;
133     input [3:0] currentmain;
134     output reg pre;
135     output reg countstart;
136     output reg draw;
137     output reg [3:0] current;
138
139     reg [3:0] next;
140
141     //declares states
142     localparam  HOLD            = 'd0,
143                 DRAW_PRE        = 'd1,
144                 DRAW_START      = 'd2,
145                 DRAW_WIN        = 'd3,
146                 DRAW_LOSE       = 'd4,
147                 DRAW_GAMEOVER   = 'd5,
148                 DRAW_PRE_HAND   = 'd6,
149                 DRAW_PRE_WAIT   = 'd7,
150                 DRAW_START_WAIT = 'd8,
151                 DRAW_WIN_WAIT   = 'd9,
152                 DRAW_LOSE_WAIT  = 'd10,
153                 DRAW_GAMEOVER_WAIT  = 'd11,
154                 DRAW_PRE_HAND_WAIT  = 'd12;
155
156     //this section dictates the order of states and
    the
157     //requirements for switching between
    states
158     always@(*)
159     begin
160     case(current)
161         HOLD: begin
162             if (currentmain == 'd0 || currentmain == 'd7
    ) next = DRAW_PRE; //If the main state is PRESTART
    or READY goes to DRAW_PRE
163             else if (start) next = DRAW_START; //if
    start is true goes to DRAW_START
164             end
165         DRAW_PRE: begin
```

```
166             if (draw_done) next = DRAW_PRE_WAIT;
    //after drawing is done goes to DRAW_PRE_WAIT state
167             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
168         end
169         DRAW_PRE_WAIT:begin
170             if (countdone) next = DRAW_PRE_HAND;
    //after 2 seconds goes to DRAW_PRE_HAND
171             else if (start) next = DRAW_START;//if
    start is true goes to DRAW_START
172             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
173         end
174         DRAW_PRE_HAND: begin
175             if(draw_done) next = DRAW_PRE_HAND_WAIT;
    //after drawing is done goes to DRAW_PRE_HAND_WAIT
    state
176             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
177         end
178         DRAW_PRE_HAND_WAIT: begin
179             if (countdone) next = DRAW_PRE;  //after 2
    seconds goes to DRAW_PRE
180             else if (start) next = DRAW_START;//if
    start is true goes to DRAW_START
181             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
182         end
183         DRAW_START: begin
184             if(draw_done) next = DRAW_START_WAIT;
    //after drawing is done goes to DRAW_START_WAIT state
185             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
186         end
187         DRAW_START_WAIT: begin
188             if(currentmain == 'd4) next = DRAW_WIN;
    //after 2 seconds goes to DRAW_WIN
189             else if (currentmain == 'd8 || currentmain
    == 'd10) next = DRAW_LOSE; //If the main state is
    LOSE or AFTERLOSE goes to DRAW_LOSE
190         end
191         DRAW_WIN: begin
```

```
192             if(draw_done) next = DRAW_WIN_WAIT; //after
    drawing is done goes to DRAW_WIN_WAIT state
193           end
194           DRAW_WIN_WAIT: begin
195             if(countdone) next = DRAW_PRE;  //after 2
    seconds goes to DRAW_PRE
196             else if (currentmain == 'd0 || currentmain
    == 'd7) next = DRAW_PRE; //If the main state is
    PRESTART or READY goes to DRAW_PRE
197             else if (start) next = DRAW_START;//if
    start is true goes to DRAW_START
198           end
199           DRAW_LOSE: begin
200             if(draw_done) next = DRAW_LOSE_WAIT;
    //after drawing is done goes to DRAW_LOSE_WAIT state
201           end
202           DRAW_LOSE_WAIT: begin
203             if(countdone) next = DRAW_GAMEOVER;
    //after 2 seconds goes to DRAW_GAMEOVER
204             else if (currentmain == 'd0 || currentmain
    == 'd7) next = DRAW_PRE; //If the main state is
    PRESTART or READY goes to DRAW_PRE
205             else if (start) next = DRAW_START; //if
    start is true goes to DRAW_START
206           end
207           DRAW_GAMEOVER: begin
208             if(draw_done) next = DRAW_GAMEOVER_WAIT;
    //after drawing is done goes to DRAW_GAMEOVER_WAIT
    state
209           end
210           DRAW_GAMEOVER_WAIT: begin
211             if (currentmain == 'd0 || currentmain == 'd7
    ) next = DRAW_PRE; //If the main state is PRESTART
    or READY goes to DRAW_PRE
212           end
213           default: next = HOLD;
214       endcase
215       end
216
217       //sets the draw and countstart values for each
    state determining when drawing happens and when to
    wait
218       //pre value is set as well and used in
    drawdatapath to reset values
219       always@(*)
220       begin
221           pre             <= 0;
```

```
222             countstart      <= 0;
223             draw            <= 1;
224         case(current)
225             HOLD: begin
226                 pre             <= 1;
227                 countstart      <= 0;
228                 draw            <= 1;
229             end
230             DRAW_PRE: begin
231                 pre             <= 0;
232                 countstart      <= 1;
233                 draw            <= 1;
234             end
235             DRAW_PRE_WAIT: begin
236                 pre             <= 0;
237                 countstart      <= 0;
238                 draw            <= 0;
239             end
240             DRAW_PRE_HAND: begin
241                 pre             <= 0;
242                 countstart      <= 0;
243                 draw            <= 1;
244             end
245             DRAW_PRE_HAND_WAIT: begin
246                 pre             <= 0;
247                 countstart      <= 1;
248                 draw            <= 0;
249             end
250             DRAW_START: begin
251                 pre             <= 0;
252                 draw            <= 1;
253             end
254             DRAW_START_WAIT: begin
255                 pre             <= 0;
256                 countstart      <= 1;
257                 draw            <= 0;
258             end
259             DRAW_WIN: begin
260                 pre             <= 0;
261                 countstart      <= 0;
262                 draw            <= 1;
263             end
264             DRAW_WIN_WAIT: begin
265                 pre             <= 0;
266                 countstart      <= 1;
267                 draw            <= 0;
268             end
```

```
269             DRAW_LOSE : begin
270                 pre             <= 0;
271                 countstart      <= 0;
272                 draw            <= 1;
273             end
274             DRAW_LOSE_WAIT : begin
275                 pre             <= 0;
276                 countstart      <= 1;
277                 draw            <= 0;
278             end
279             DRAW_GAMEOVER : begin
280                 pre             <= 0;
281                 countstart      <= 0;
282                 draw            <= 1;
283             end
284             DRAW_GAMEOVER_WAIT : begin
285                 pre             <= 0;
286                 countstart      <= 1;
287                 draw            <= 0;
288             end
289         endcase
290         end
291
292         //sets state transition to clock edge
293         always@(posedge clk)
294         begin
295         if(!resetn)
296             current = HOLD;
297         else
298             current = next;
299         end
300
301     endmodule
302
303
304     //the drawdatapath is used to set the draw location
        and give it a colour
305     module drawdatapath( clk,
306                          resetn,
307                          pre,
308                          draw,
309                          imagedata,
310                          numberdata1,
311                          numberdata2,
312                          numberdata3,
313                          colour_out,
314                          x_out,
```

```
315                         y_out,
316                         address,
317                         draw_done,
318                         x_counter,
319                         y_counter
320                         );
321
322     input clk;
323     input resetn;
324     input pre;
325     input draw;
326     input [23:0] imagedata;
327     input [23:0] numberdata1;
328     input [23:0] numberdata2;
329     input [23:0] numberdata3;
330     output reg [23:0] colour_out;
331     output reg [7:0] x_out;
332     output reg [6:0] y_out;
333     output reg [16:0] address;
334     output reg draw_done;
335     output reg [7:0] x_counter;
336     output reg [6:0] y_counter;
337
338     //x and y denote the start position for each
    destinct drawing cell
339     reg [7:0] x;
340     reg [6:0] y;
341
342     always@(posedge clk)
343     begin
344
345     //combines start positions and their respective
    counters
346     x_out <= x + x_counter;
347     y_out <= y + y_counter;
348
349         //resets values to 0 when pre is true
350         if(pre) begin
351             x <= 0;
352             y <= 0;
353             x_counter <= 0;
354             y_counter <= 0;
355             x_out <= 0;
356             y_out <= 0;
357             draw_done <= 0;
358         end
359
```

```
360         if(draw) begin
361
362             //sets colour output for the first 20 rows
    with the last 3 20x20 square displaying the score
363             if(x == 0 && y == 0) colour_out <= 'b0;
364             else if(x == 20 && y == 0) colour_out <= 'b0;
365             else if(x == 40 && y == 0) colour_out <= 'b0;
366             else if(x == 60 && y == 0) colour_out <= 'b0;
367             else if(x == 80 && y == 0) colour_out <= 'b0;
368             else if(x == 100 && y == 0) colour_out <=
    numberdata3;
369             else if(x == 120 && y == 0) colour_out <=
    numberdata2;
370             else if(x == 140 && y == 0) colour_out <=
    numberdata1;
371             //sets colour output for the rest of the
    display
372             else if(y == 20) begin
373                 colour_out <= imagedata;
374             end
375
376             //increments draw location through the
    first 20 rows
377             //it increments x 20 pixels then increments
    y by 1
378             //when y and x get to the bottom right
    corner of the 20x20 square
379             //it moves to the top left of the next square
380             //after the row of squares is done y is set
    to 20 and the main image is drawn
381             if(y == 0) begin
382                 if(x_counter < 18) begin
383                     x_counter <= x_counter + 1;
384                 end
385                 else if(x_counter == 18) begin
386                     x_counter <= 0;
387                     if(y_counter < 19) y_counter <=
    y_counter + 1;
388                     else begin
389                         x_counter <= 0;
390                         y_counter <= 0;
391                         if(x < 160) x <= x + 20;
392                         else begin
393                             y<=20;
394                             x<=0;
395                             x_counter <= 0;
396                             y_counter <= 0;
```

```
397                     end
398                 end
399             end
400         end
401
402
403         //when y == 20 the drawn pixels are incremented across the display and at the end moved down by one
404         else if(y == 20) begin
405             x <= 0;
406             y <= 20;
407             if(x_counter < 159) begin
408                 x_counter <= x_counter + 1;
409             end
410             else if(x_counter == 159) begin
411                 x_counter <= 0;
412                 if(y_counter < 100) y_counter <= y_counter + 1;
413                 else begin
414                     draw_done <= 1;
415                     x <= 0;
416                     y <= 0;
417                 end
418             end
419
420         end
421
422
423
424     end
425     else begin
426         x <= 0;
427         y <= 0;
428         x_counter <= 0;
429         y_counter <= 0;
430         x_out <= 0;
431         y_out <= 0;
432         draw_done <= 0;
433     end
434 end
435
436
437 endmodule
438
439 //selects the starting address to draw the main image base on the FSM output
```

```
440   module backSelect(back_select, address_back);
441      input [3:0] back_select;
442      output reg [17:0] address_back;
443
444      always@(*)
445      begin
446         case(back_select)
447            4'd0     :  address_back = 'd0;
      //HOLD
448            4'd1     :  address_back = 'd0;
      //PRE
449            4'd2     :  address_back = 'd16000;
      //START
450            4'd3     :  address_back = 'd32000;
      //WIN
451            4'd4     :  address_back = 'd48000;
      //LOSE
452            4'd5     :  address_back = 'd64000;
      //GAMEOVER
453            4'd6     :  address_back = 'd16000;
      //DRAW_PRE_HAND
454            4'd7     :  address_back = 'd0;
      //PRE
455            4'd8     :  address_back = 'd16000;
      //START
456            4'd9     :  address_back = 'd32000;
      //WIN
457            4'd10    :  address_back = 'd48000;
      //LOSE
458            4'd11    :  address_back = 'd64000;
      //GAMEOVER
459            4'd12    :  address_back = 'd16000;
      //DRAW_PRE_HAND
460            default  :  address_back = 'd0;
      //0
461         endcase
462      end
463
464      endmodule
465
466   //selects the starting address to draw the number
      for each score digit based
467   //on score inputs from the main game datapath
468   module numberSelect(num_select, address_num);
469      input [4:0] num_select;
470      output reg [11:0] address_num;
471
```

```
472         always@(*)
473         begin
474            case(num_select)
475               4'b0000      :  address_num = 'd0;
                     //0
476               4'b0001      :  address_num = 'd400;
                   //1
477               4'b0010      :  address_num = 'd800;
                   //2
478               4'b0011      :  address_num = 'd1200;
                  //3
479               4'b0100      :  address_num = 'd1600;
                  //4
480               4'b0101      :  address_num = 'd2000;
                  //5
481               4'b0110      :  address_num = 'd2400;
                  //6
482               4'b0111      :  address_num = 'd2800;
                  //7
483               4'b1000      :  address_num = 'd3200;
                  //8
484               4'b1001      :  address_num = 'd3600;
                  //9
485               default   :  address_num = 'd0;
      //0
486            endcase
487         end
488
489         endmodule
490
491         //2 second countdown clock
492      module countdown_2(clk, loadEnable, countDone);
493          input clk, loadEnable;
494          output reg countDone;
495
496          reg [32:0]countVal;
497
498         always @(posedge clk) begin
499         if (!loadEnable) begin
500            countVal <= 'd100000000;
501            countDone <= 0;
502         end
503
504         else if (countVal == 'd0) begin
505               countDone <= 1;
506         end
507
```

```
508         else if(countVal != 'd0) begin
509             countVal <= countVal - 1;
510             countDone <= 0;
511         end
512         end
513 
514     endmodule
```

```
1   //*********************************TOP
    MODULE*************************************************
    *********//
2
3   //-----variable
    names::-------------------------------------------
    ----------------------------------//
4   //open_fsm and close_fsm are signals to open and
    close coming from the central game control
5   //limit_jawOpen: limit switch that indicates if jaw
    is open when depressed (similar for limit_jawClose)
6   //jaw_is_open and jaw_is_closed are "handshakes"
    back to the main game control
7
8   //---potential
    bugs:---------------------------------------------
    ----------------------------------//
9   //counter that is used to slow the motor
10  //signal may not get thru to the FPGA since there is
    no initial value for countVal
11
12  module motorFSM(clock, resetn, open_fsm, close_fsm,
    limit_jawOpen, limit_jawClose, jaw_is_open,
    jaw_is_closed, pin, current);
13
14      input clock, resetn, open_fsm, close_fsm,
    limit_jawOpen, limit_jawClose;
15      output jaw_is_open, jaw_is_closed;
16      output [3:0]pin;
17
18      wire openJaw, closeJaw, counterClock;
19      output [3:0] current;
20
21      //everytime the countDone == 1 --> rising edge
    of the counterClock
22      //counterClock used to slow down the motor
23      countdown_motor countdown_motor(.clock(clock), .
    countDone(counterClock));
24
25      controlMotor controlMotor1(.clock(clock),
26                                 .resetn(resetn),
27                                  .open_fsm(open_fsm),
28                                  .close_fsm(close_fsm),
29                                  .limit_jawOpen(
    limit_jawOpen),
30                                  .limit_jawClose(
    limit_jawClose),
```

```
31                                          .openJaw(openJaw),
32                                          .closeJaw(closeJaw),
33                                          .jaw_is_open(
    jaw_is_open),
34                                          .jaw_is_closed(
    jaw_is_closed),
35                                          .current(current));
36
37
38      datapathMotor datapathMotor1(.counterClock(
    counterClock),
39                                          .resetn(resetn),
40                                          .openJaw(openJaw),
41                                          .closeJaw(closeJaw),
42                                          .pin(pin));
43  endmodule
44
45
46
47
48
49
50
51
52
53
54
55
56
57  //reset will be connected to the reset that exists
    in all of the modules
58  //reset doesn't do anything...mostly for modelsim
    purposes
59
60  module controlMotor(clock, resetn, open_fsm,
    close_fsm, limit_jawOpen, limit_jawClose, openJaw,
    closeJaw, jaw_is_open, jaw_is_closed, current) ;
61
62      input clock, resetn, open_fsm, close_fsm;
    //open_fsm and close_fsm: signals from central game
    control that tells this module what to do
63      input limit_jawOpen, limit_jawClose; //limit
    switches to detect if jaw is open or closed
64
65      output reg openJaw, closeJaw; //to control the
    motor
66      output reg jaw_is_closed, jaw_is_open;
```

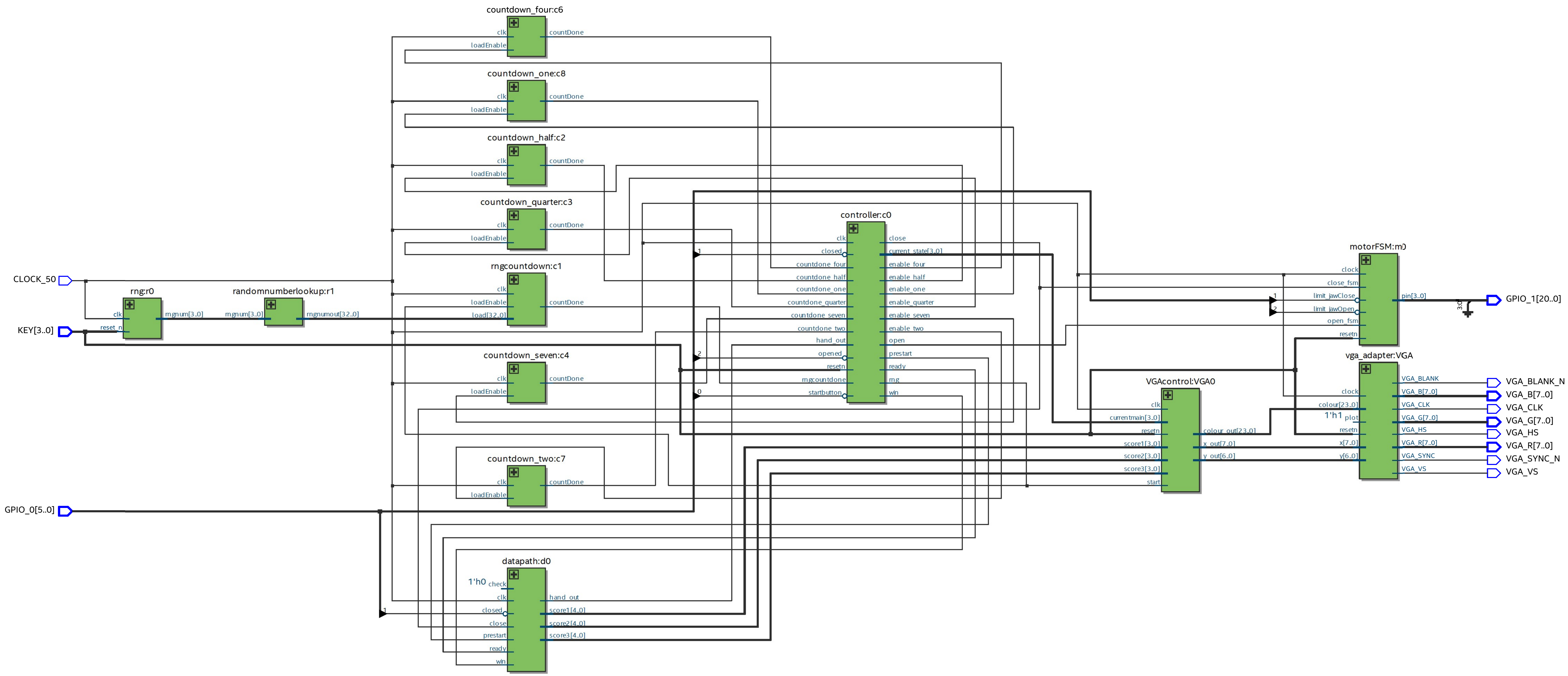
```
    //"handshake" for the central game control
67
68      //state registers
69      output reg[3:0] current;
70      reg [3:0] next;
71
72      //hold state is to allow time for the central
    game control to check game status (aka if hand is
    caught or not)
73      //should recieve signal from central game
    control to open back up the jaws
74      localparam READY = 4'd0,
75                 CLOSE = 4'd1,
76                 HOLD  = 4'd2,
77                 OPEN  = 4'd3;
78
79      //state table
80      always @(*)
81      begin: state_table
82        case(current)
83          READY: begin
84            if (close_fsm) next = CLOSE;  //goes to
    CLOSE state if close signal recieved
85            else if(open_fsm) next = OPEN; //goes to
    OPEN state if open signal recieved
86          end
87          CLOSE: begin
88            if (limit_jawClose) next = READY;
    //returns to READY if when jaw is closed
89            else if(open_fsm) next = OPEN;   //goes
    to OPEN if open signal recieved
90          end
91          OPEN: begin
92            if (limit_jawOpen) next = READY;
    //returns to READY if when jaw is opened
93          end
94          default: next = READY;
95        endcase
96      end
97
98      //datapath controls
99      always @ (*)
100     begin: enable_signals
101       closeJaw = 'd0;
102       openJaw = 'd0;
103       jaw_is_closed = 'd0;
104       jaw_is_open = 'd0;
```

```
105
106             case (current)
107                 READY: begin
108                     closeJaw = 'd0;
109                     openJaw = 'd0;
110                     jaw_is_closed = 'd0;
111                     jaw_is_open = 'd1;
112                 end
113
114                 CLOSE: begin
115                     closeJaw = 'd1;
116                     openJaw = 'd0;
117                     jaw_is_closed = 'd0;
118                     jaw_is_open = 'd0;
119                 end
120
121                 HOLD: begin
122                     closeJaw = 'd0;
123                     openJaw = 'd0;
124                     jaw_is_closed = 'd1;
125                     jaw_is_open = 'd0;
126                 end
127
128                 OPEN: begin
129                     closeJaw = 'd0;
130                     openJaw = 'd1;
131                     jaw_is_closed = 'd0;
132                     jaw_is_open = 'd0;
133                 end
134             endcase
135         end
136
137         //current state registers
138         always @ (posedge clock)
139             begin
140                 if (!resetn) current <=READY;
141                 else current <= next;
142             end
143
144     endmodule
145
146
147     //there will be a feedback loop to see which pins
        were last triggered
148     module datapathMotor(counterClock, resetn, openJaw,
        closeJaw, pin);
149
```

```
150         input counterClock, resetn, openJaw, closeJaw;
151         output reg [3:0]pin;
152 
153         reg [3:0]even_odd;
154 
155         // //output to pins also stored in wires
156         // wire [3:0]p;
157 
158         always @ (posedge counterClock) begin
159             even_odd <= even_odd + 1;
160 
161             if (!resetn) begin
162                 pin[3:0] <= 'd0;
163             end
164 
165 
166 
167             else if (closeJaw) begin
168                 case (pin[3:0])
169                     4'b1100: pin = 4'b0110;
170                     4'b0110: pin = 4'b0011;
171                     4'b0011: pin = 4'b1001;
172                     4'b1001: pin = 4'b1100;
173                     default: pin = 4'b1100;
174                 endcase
175             end
176 
177 
178 
179             else if (openJaw) begin
180                 if(even_odd == 0 || even_odd == 3 ||
    even_odd == 6 || even_odd == 9 || even_odd == 12)
    begin
181                     case (pin[3:0])
182                         4'b0011: pin = 4'b0110;
183                         4'b0110: pin = 4'b1100;
184                         4'b1100: pin = 4'b1001;
185                         4'b1001: pin = 4'b0011;
186                         default: pin = 4'b0011;
187                     endcase
188                end
189             end
190 
191             //when the motor shouldnt be stimulated
192             else begin
193                 case (pin[3:0])
194                     4'b1100: pin = 4'b0000;
```

```
195                     4'b0110: pin = 4'b0000;
196                     4'b0011: pin = 4'b0000;
197                     4'b1001: pin = 4'b0000;
198                     default: pin = 4'b0000;
199                 endcase
200             end
201
202         end
203     endmodule
204
205     //motor clock....values need to be changed
206     //potential bugs in here...not sure if resetn is
        needed? **********************
207     module countdown_motor(clock, countDone);
208         input clock;
209         output reg countDone;
210
211         reg [32:0]countVal;
212
213         always @(posedge clock) begin
214             if (countVal == 'd0) begin
215                 countVal <= 'd100000; //change values as
        needed
216                 countDone <= 1;
217             end
218
219             else if(countVal != 'd0) begin
220                countVal <= countVal - 1;
221                countDone <= 0;
222             end
223         end
224     endmodule
225
```

Appendix B - full block diagram of game implementation


countdown_four:c6
clk
loadEnable
countDone
countdown_one:c8
clk
loadEnable
countDone
countdown_half:c2
clk
loadEnable
countDone
countdown_quarter:c3
clk
loadEnable
countDone
rngcountdown:c1
clk
loadEnable
load[32..0]
countDone
countdown_seven:c4
clk
loadEnable
countDone
countdown_two:c7
clk
loadEnable
countDone
CLOCK_50
KEY[3..0]
GPIO_0[5..0]
rng:r0
clk
reset_n
rngnum[3..0]
randomnumberlookup:r1
rngnum[3..0]
rngnumout[32..0]
controller:c0
clk
closed
countdone_four
countdone_half
countdone_one
countdone_quarter
countdone_seven
countdone_two
hand_out
opened
resetn
rngcountdone
startbutton
close
current_state[3..0]
enable_four
enable_half
enable_one
enable_quarter
enable_seven
enable_two
open
prestart
ready
rng
win
datapath:d0
1'h0
check
clk
closed
close
prestart
ready
win
hand_out
score1[4..0]
score2[4..0]
score3[4..0]
motorFSM:m0
clock
close_fsm
limit_jawClose
limit_jawOpen
open_fsm
resetn
pin[3..0]
3:0
GPIO_1[20..0]
VGAcontrol:VGA0
clk
currentmain[3..0]
resetn
score1[3..0]
score2[3..0]
score3[3..0]
start
colour_out[23..0]
x_out[7..0]
y_out[6..0]
vga_adapter:VGA
clock
colour[23..0]
1'h1
plot
resetn
x[7..0]
y[6..0]
VGA_BLANK
VGA_B[7..0]
VGA_CLK
VGA_G[7..0]
VGA_HS
VGA_R[7..0]
VGA_SYNC
VGA_VS
VGA_BLANK_N
VGA_B[7..0]
VGA_CLK
VGA_G[7..0]
VGA_HS
VGA_R[7..0]
VGA_SYNC_N
VGA_VS

Appendix C - Mechanical jaw system design. Sketches were done in SOLIDWORKS and exported onto Adobe Illustrator prior to being laser cut. Dimensions shown are represented in inches.