

```

1  /*****
2
3  animator.sv
4
5  *****/
6
7  /*
8   * Connor Aksama
9   * 01/30/2023
10  * CSE 371
11  * Lab 3
12  */
13
14  /**
15   * Outputs the endpoints of the current line to draw on the display and a signal
16   * indicating whether
17   * the screen should be cleared.
18
19   * Inputs:
20   *   clk [1 bit] - the clock to use for this module
21   *   reset [1 bit] - the reset signal for this module; resets to the first frame when
22   *   raised
23
24   * Outputs:
25   *   x0 [10 bit] - The x coordinate of the first point of the line to draw
26   *   y0 [9 bit] - The y coordinate of the first point of the line to draw
27   *   x1 [10 bit] - The x coordinate of the second point of the line to draw
28   *   y1 [9 bit] - The y coordinate of the second point of the line to draw
29   *   clear [1 bit] - Signal to indicate when the screen should be cleared
30  */
31  module animator#(
32      parameter frame_time = 833333
33  ) (
34      output logic [9:0] x0, x1
35      ,output logic [8:0] y0, y1
36      ,output logic clear
37      ,input logic clk, reset
38  );
39
40      logic next_frame, next_frame_q;
41      logic [31:0] frame_num;
42
43      // Counter to time the duration of each frame
44      // timer_out -> number of cycles current frame is active
45      // inc on every cycle; no dec; use same clock, reset on frame switch/reset
46      logic [31:0] timer_out;
47      counter_32b timer (.count(timer_out), .inc(1'b1), .dec(1'b0), .clk, .reset(reset |
48      next_frame));
49
50      // Counter to keep track of the current frame number
51      // frame_num -> current frame number
52      // inc on every frame switch; no dec; use same clock, reset on end of sequence/reset
53      logic frame_reset;
54      counter_32b frame_counter (.count(frame_num), .inc(next_frame_q), .dec(1'b0), .clk, .
55      reset(reset | frame_reset));
56
57      // Determine if timer expired, switch to next frame logic
58      always_comb begin
59
60          // Check for next frame
61          next_frame = (timer_out >= frame_time - 1) && !next_frame_q;
62
63          // Clear screen every other frame
64          clear = ~frame_num[0];
65
66          // End of sequence reached
67          frame_reset = (frame_num >= 100);
68
69          // Frames

```

```

66     case (frame_num[31:1])
67
68         0:begin x0=420;y0=240;x1=220;y1=240;end 1:begin x0=419;y0=243;x1=221;y1=237;
end 2:begin x0=419;y0=246;x1=221;y1=234;end 3:begin x0=419;y0=249;x1=221;y1
=231;end
69 4:begin x0=419;y0=252;x1=221;y1=228;end 5:begin x0=418;y0=255;x1=222;y1=225;
end 6:begin x0=418;y0=258;x1=222;y1=222;end 7:begin x0=417;y0=261;x1=223;y1
=219;end
70 8:begin x0=416;y0=264;x1=224;y1=216;end 9:begin x0=416;y0=267;x1=224;y1=213;
end 10:begin x0=415;y0=270;x1=225;y1=210;end 11:begin x0=414;y0=273;x1=226;
y1=207;end
71 12:begin x0=412;y0=276;x1=228;y1=204;end 13:begin x0=411;y0=279;x1=229;y1=
201;end 14:begin x0=410;y0=282;x1=230;y1=198;end 15:begin x0=409;y0=285;x1=
231;y1=195;end
72 16:begin x0=407;y0=288;x1=233;y1=192;end 17:begin x0=406;y0=290;x1=234;y1=
190;end 18:begin x0=404;y0=293;x1=236;y1=187;end 19:begin x0=402;y0=296;x1=
238;y1=184;end
73 20:begin x0=400;y0=298;x1=240;y1=182;end 21:begin x0=399;y0=301;x1=241;y1=
179;end 22:begin x0=397;y0=303;x1=243;y1=177;end 23:begin x0=395;y0=306;x1=
245;y1=174;end
74 24:begin x0=392;y0=308;x1=248;y1=172;end 25:begin x0=390;y0=310;x1=250;y1=
170;end 26:begin x0=388;y0=312;x1=252;y1=168;end 27:begin x0=386;y0=315;x1=
254;y1=165;end
75 28:begin x0=383;y0=317;x1=257;y1=163;end 29:begin x0=381;y0=319;x1=259;y1=
161;end 30:begin x0=378;y0=320;x1=262;y1=160;end 31:begin x0=376;y0=322;x1=
264;y1=158;end
76 32:begin x0=373;y0=324;x1=267;y1=156;end 33:begin x0=370;y0=326;x1=270;y1=
154;end 34:begin x0=368;y0=327;x1=272;y1=153;end 35:begin x0=365;y0=329;x1=
275;y1=151;end
77 36:begin x0=362;y0=330;x1=278;y1=150;end 37:begin x0=359;y0=331;x1=281;y1=
149;end 38:begin x0=356;y0=332;x1=284;y1=148;end 39:begin x0=353;y0=334;x1=
287;y1=146;end
78 40:begin x0=350;y0=335;x1=290;y1=145;end 41:begin x0=347;y0=336;x1=293;y1=
144;end 42:begin x0=344;y0=336;x1=296;y1=144;end 43:begin x0=341;y0=337;x1=
299;y1=143;end
79 44:begin x0=338;y0=338;x1=302;y1=142;end 45:begin x0=335;y0=338;x1=305;y1=
142;end 46:begin x0=332;y0=339;x1=308;y1=141;end 47:begin x0=329;y0=339;x1=
311;y1=141;end
80 48:begin x0=326;y0=339;x1=314;y1=141;end 49:begin x0=323;y0=339;x1=317;y1=
141;end 50:begin x0=320;y0=340;x1=320;y1=140;end 51:begin x0=317;y0=339;x1=
323;y1=141;end
81 52:begin x0=314;y0=339;x1=326;y1=141;end 53:begin x0=311;y0=339;x1=329;y1=
141;end 54:begin x0=308;y0=339;x1=332;y1=141;end 55:begin x0=305;y0=338;x1=
335;y1=142;end
82 56:begin x0=302;y0=338;x1=338;y1=142;end 57:begin x0=299;y0=337;x1=341;y1=
143;end 58:begin x0=296;y0=336;x1=344;y1=144;end 59:begin x0=293;y0=336;x1=
347;y1=144;end
83 60:begin x0=290;y0=335;x1=350;y1=145;end 61:begin x0=287;y0=334;x1=353;y1=
146;end 62:begin x0=284;y0=332;x1=356;y1=148;end 63:begin x0=281;y0=331;x1=
359;y1=149;end
84 64:begin x0=278;y0=330;x1=362;y1=150;end 65:begin x0=275;y0=329;x1=365;y1=
151;end 66:begin x0=272;y0=327;x1=368;y1=153;end 67:begin x0=270;y0=326;x1=
370;y1=154;end
85 68:begin x0=267;y0=324;x1=373;y1=156;end 69:begin x0=264;y0=322;x1=376;y1=
158;end 70:begin x0=262;y0=320;x1=378;y1=160;end 71:begin x0=259;y0=319;x1=
381;y1=161;end
86 72:begin x0=257;y0=317;x1=383;y1=163;end 73:begin x0=254;y0=315;x1=386;y1=
165;end 74:begin x0=252;y0=312;x1=388;y1=168;end 75:begin x0=250;y0=310;x1=
390;y1=170;end
87 76:begin x0=248;y0=308;x1=392;y1=172;end 77:begin x0=245;y0=306;x1=395;y1=
174;end 78:begin x0=243;y0=303;x1=397;y1=177;end 79:begin x0=241;y0=301;x1=
399;y1=179;end
88 80:begin x0=240;y0=298;x1=400;y1=182;end 81:begin x0=238;y0=296;x1=402;y1=
184;end 82:begin x0=236;y0=293;x1=404;y1=187;end 83:begin x0=234;y0=290;x1=
406;y1=190;end
89 84:begin x0=233;y0=288;x1=407;y1=192;end 85:begin x0=231;y0=285;x1=409;y1=
195;end 86:begin x0=230;y0=282;x1=410;y1=198;end 87:begin x0=229;y0=279;x1=
411;y1=201;end
90 88:begin x0=228;y0=276;x1=412;y1=204;end 89:begin x0=226;y0=273;x1=414;y1=

```

```

207;end 90:begin x0=225;y0=270;x1=415;y1=210;end 91:begin x0=224;y0=267;x1=
416;y1=213;end
91 92:begin x0=224;y0=264;x1=416;y1=216;end 93:begin x0=223;y0=261;x1=417;y1=
219;end 94:begin x0=222;y0=258;x1=418;y1=222;end 95:begin x0=222;y0=255;x1=
418;y1=225;end
92 96:begin x0=221;y0=252;x1=419;y1=228;end 97:begin x0=221;y0=249;x1=419;y1=
231;end 98:begin x0=221;y0=246;x1=419;y1=234;end 99:begin x0=221;y0=243;x1=
419;y1=237;end
93 default: begin x0=0;y0=0;x1=0;y1=0; end
94
95     endcase
96
97 end
98
99 // Register the next frame signal to frame counter
100 always_ff @(posedge clk) begin
101     if (reset) begin
102         next_frame_q <= 0;
103     end else begin
104         next_frame_q <= next_frame;
105     end
106 end
107
108 endmodule // animator
109
110 // Module to test the functionality of the animator module
111 module animator_testbench();
112
113     logic [9:0] x0, x1;
114     logic [8:0] y0, y1;
115     logic clear;
116     logic clk, reset;
117
118     animator #(2) dut (.*) ;
119
120     parameter CLOCK_PERIOD = 100;
121     initial begin
122         clk <= 0;
123         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
124     end
125
126     initial begin
127         integer i;
128
129         @(posedge clk) reset <= 1;
130         @(posedge clk) reset <= 0;
131
132         // Run through each of the frames in the animation
133         for (i = 0; i < 500; i++) @(posedge clk);
134
135         $stop;
136     end
137
138
139 endmodule // animator_testbench
140
141 /*****
142
143 counter_32b.sv
144
145 *****/
146
147 /*
148  * Connor Aksama
149  * 01/30/2023
150  * CSE 371
151  * Lab 3
152  */
153

```

```

154  /**
155  * A thirty two-bit increment/decrement counter. Overflows to 8'h0; underflows to
156  * 8'hFFFF_FFFF.
157  * reset takes precedence; if inc and dec are simultaneously raised, count is unchanged.
158  * Inputs:
159  *   inc [1 bit] - Increments the count by 1 when raised. Does nothing if signal is low.
160  *   dec [1 bit] - Decrements the count by 1 when raised. Does nothing if signal is low.
161  *   clk [1 bit] - The clock to use for this module.
162  *   reset [1 bit] - Resets the counter to 0 when raised. Does nothing if signal is low.
163
164  * Outputs:
165  *   count [32 bit] - The count of this counter.
166  */
167 module counter_32b(
168     output logic [31:0] count
169     ,input logic inc, dec, clk, reset
170 );
171
172     // Handles increment/decrement/reset operations
173     always_ff @(posedge clk) begin
174
175         if (reset)
176             count <= 32'b0;
177         else if (inc & ~dec)
178             count <= count + 32'b1;
179         else if (dec & ~inc)
180             count <= count - 32'b1;
181
182     end
183 endmodule // counter_32b
184
185 // Module to test the functionality of the counter_32b module
186 module counter_32b_testbench();
187
188     logic [31:0] count;
189     logic inc, dec, clk, reset;
190
191     counter_32b dut (.*);
192
193     parameter CLOCK_PERIOD = 100;
194     initial begin
195         clk <= 0;
196         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
197     end
198
199     initial begin
200         integer i;
201
202         @(posedge clk) reset <= 1;
203         @(posedge clk) reset <= 0; inc <= 1; dec <= 0;
204         for (i = 0; i < 40; i++) begin
205             @(posedge clk);
206         end
207
208         @(posedge clk) reset <= 0; inc <= 0; dec <= 1;
209         for (i = 0; i < 50; i++) begin
210             @(posedge clk);
211         end
212
213         @(posedge clk) reset <= 0; inc <= 1; dec <= 0;
214         for (i = 0; i < 15; i++) begin
215             @(posedge clk);
216         end
217
218         $stop;
219     end
220 endmodule // counter_32b_testbench

```

```

222
223 /*****
224
225 DE1_SoC.sv
226
227 *****/
228
229 /*
230  * Connor Aksama
231  * 01/30/2023
232  * CSE 371
233  * Lab 3
234 */
235
236 /**
237  * Top-level module for Lab 3. Instantiates drawing logic modules and connections to VGA
  controller
238
239  * Inputs:
240  * SW [10 bit] - The 10 onboard switches, respectively.
241  * KEY [4 bit] - The 4 onboard keys, respectively.
242  * CLOCK_50 [1 bit] - The system clock to use for this module.
243
244  * Outputs:
245  * HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
  display format.
246  * HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
  display format.
247  * HEX2 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
  display format.
248  * HEX3 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
  display format.
249  * HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
  display format.
250  * HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
  display format.
251  * LEDR [10 bit] - Signal to output to 10 onboard LEDs, respectively.
252  * VGA_R [8 bit] - The red channel value of the pixel to write to the display
253  * VGA_G [8 bit] - The green channel value of the pixel to write to the display
254  * VGA_B [8 bit] - The blue channel value of the pixel to write to the display
255  * VGA_BLANK_N [1 bit] - Blank area signal
256  * VGA_CLK [1 bit] - Divided clock for VGA signal
257  * VGA_HS [1 bit] - VGA timing signal
258  * VGA_SYNC_N [1 bit] - Unused
259  * VGA_VS [1 bit] - VGA timing signal
260 */
261 module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
262   VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);
263
264   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
265   output logic [9:0] LEDR;
266   input logic [3:0] KEY;
267   input logic [9:0] SW;
268
269   input CLOCK_50;
270   output [7:0] VGA_R;
271   output [7:0] VGA_G;
272   output [7:0] VGA_B;
273   output VGA_BLANK_N;
274   output VGA_CLK;
275   output VGA_HS;
276   output VGA_SYNC_N;
277   output VGA_VS;
278
279   assign HEX1 = '1;
280   assign HEX0 = '1;
281   assign HEX2 = '1;
282   assign HEX3 = '1;
283   assign HEX4 = '1;

```

```

284 assign HEX5 = '1;
285 assign LEDR = SW;
286
287 // Outputs from drawing modules
288 logic [9:0] x0_l, x0_a, x1, x1_l, x1_a;
289 logic [8:0] y0_l, y0_a, y1, y1_l, y1_a;
290 // Inputs into frame buffer
291 logic [9:0] x_l, x_a, x_c, x;
292 logic [8:0] y_l, y_a, y_c, y;
293 logic frame_start;
294 logic pixel_color;
295 logic reset, clear_screen;
296
297 assign reset = SW[9];
298 assign clear_screen = SW[8];
299
300
301 //////////////////////////////////////////////////
302 logic dfb_en;
303 assign dfb_en = 1'b0;
304 //////////////////////////////////////////////////
305
306 VGA_framebuffer fb(.clk(CLOCK_50), .rst(1'b0), .x, .y,
307                  .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
308                  .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
309                  .VGA_BLANK_N, .VGA_SYNC_N);
310
311 // Line Drawing Module
312 // Inputs -> system clock, reset signal, (x0_l, y0_l), (x1_l, y1_l) -> from local
313 // temp logic; defines endpoints of line
314 // Outputs -> (x_l, y_l) -> pixel to draw to VGA - changes by at most one pixel each
315 // clock cycle, starting one
316 // cycle after input points change
317 line_drawer lines (.clk(CLOCK_50), .reset,
318                  .x0(x0_l), .y0(y0_l), .x1(x1_l), .y1(y1_l), .x(x_l), .y(y_l));
319
320 // Animator Module
321 // Inputs -> system clock, reset signal
322 // Outputs -> (x0_a, y0_a), (x1_a, y1_a) -> Endpoints of line to draw
323 // clear -> signal is high when screen should be cleared
324 logic clear;
325 animator #(50000000) anim (.x0(x0_a), .x1(x1_a), .y0(y0_a), .y1(y1_a), .clear, .clk(
326 CLOCK_50), .reset);
327
328 // Screen Clearing Module; cycles through each pixel on screen
329 // Inputs -> system clock, reset signal
330 // Output -> (x_c, y_c) -> Coordinate of pixel to clear on display
331 fill_space #(640,480) clearer (.x(x_c), .y(y_c), .clk(CLOCK_50), .reset);
332
333 // Multiplex endpoints into line drawing module
334 // Multiplex coordinates into frame buffer
335 always_comb begin
336     if (SW[7]) begin
337         // Display arbitrary lines based on SW input
338         case (SW[2:0])
339             0: begin
340                 // vertical
341                 x0_l = 50;
342                 y0_l = 50;
343                 x1_l = 50;
344                 y1_l = 400;
345                 pixel_color = 1'b1;
346             end
347             1: begin
348                 // horizontal
349                 x0_l = 90;
350                 y0_l = 90;
351                 x1_l = 450;
352                 y1_l = 90;

```

```

350         pixel_color = 1'b1;
351     end
352     2: begin
353         // negative diagonal
354         x0_l = 240;
355         y0_l = 240;
356         x1_l = 0;
357         y1_l = 0;
358         pixel_color = 1'b1;
359     end
360     3: begin
361         // positive diagonal
362         x0_l = 400;
363         y0_l = 40;
364         x1_l = 40;
365         y1_l = 400;
366         pixel_color = 1'b1;
367     end
368     4: begin
369         // positive shallow
370         x0_l = 400;
371         y0_l = 40;
372         x1_l = 40;
373         y1_l = 100;
374         pixel_color = 1'b1;
375     end
376     5: begin
377         // negative shallow
378         x0_l = 300;
379         y0_l = 200;
380         x1_l = 40;
381         y1_l = 100;
382         pixel_color = 1'b1;
383     end
384     6: begin
385         // positive steep
386         x0_l = 40;
387         y0_l = 400;
388         x1_l = 80;
389         y1_l = 40;
390         pixel_color = 1'b1;
391     end
392     7: begin
393         // negative steep
394         x0_l = 40;
395         y0_l = 40;
396         x1_l = 80;
397         y1_l = 400;
398         pixel_color = 1'b1;
399     end
400     default: begin
401         x0_l = 0;
402         y0_l = 0;
403         x1_l = 0;
404         y1_l = 0;
405         pixel_color = 1'b1;
406     end
407
408 endcase
409 x = x_l;
410 y = y_l;
411 end
412 else if (clear | clear_screen) begin
413     // Clear Screen
414     x0_l = 0;
415     y0_l = 0;
416     x1_l = 0;
417     y1_l = 0;
418     pixel_color = 1'b0;

```

```

419         x = x_c;
420         y = y_c;
421     end
422     else begin
423         // Display animation
424         x0_l = x0_a;
425         y0_l = y0_a;
426         x1_l = x1_a;
427         y1_l = y1_a;
428         pixel_color = 1'b1;
429         x = x_l;
430         y = y_l;
431     end
432 end
433
434
435 endmodule // DE1_SoC
436
437 // Module to test the functionality of the DE1_SoC module
438 module DE1_SoC_testbench();
439     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
440     logic [9:0] LEDR;
441     logic [3:0] KEY;
442     logic [9:0] SW;
443
444     logic clk;
445     logic [7:0] VGA_R;
446     logic [7:0] VGA_G;
447     logic [7:0] VGA_B;
448     logic VGA_BLANK_N;
449     logic VGA_CLK;
450     logic VGA_HS;
451     logic VGA_SYNC_N;
452     logic VGA_VS;
453
454     DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5
455                 ,.LEDR, .KEY, .SW, .CLOCK_50(clk)
456                 ,.VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N
457                 ,.VGA_CLK, .VGA_HS, .VGA_SYNC_N, .VGA_VS);
458
459     parameter CLOCK_PERIOD = 100;
460     initial begin
461         clk <= 0;
462         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
463     end
464
465     initial begin
466         integer i;
467
468         @(posedge clk) SW[9] <= 1;
469         @(posedge clk) SW[9] <= 0;
470         @(posedge clk) SW[7] <= 0;
471
472         for (i = 0; i < 50; i++) begin
473             @(posedge clk);
474         end
475
476         // Run through each of the arbitrary lines
477         @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b000;
478         for (i = 0; i < 10; i++) begin
479             @(posedge clk);
480         end
481         @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b001;
482         for (i = 0; i < 10; i++) begin
483             @(posedge clk);
484         end
485         @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b010;
486         for (i = 0; i < 10; i++) begin
487             @(posedge clk);

```



```

488     end
489     @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b011;
490     for (i = 0; i < 10; i++) begin
491         @(posedge clk);
492     end
493     @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b100;
494     for (i = 0; i < 10; i++) begin
495         @(posedge clk);
496     end
497     @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b101;
498     for (i = 0; i < 10; i++) begin
499         @(posedge clk);
500     end
501     @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b110;
502     for (i = 0; i < 10; i++) begin
503         @(posedge clk);
504     end
505     @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b111;
506     for (i = 0; i < 10; i++) begin
507         @(posedge clk);
508     end
509     // Clear the screen
510     @(posedge clk) SW[7] <= 0; SW[8] <= 1;
511     for (i = 0; i < 20; i++) begin
512         @(posedge clk);
513     end
514     // Run through the animation
515     @(posedge clk) SW[8] <= 0;
516     for (i = 0; i < 100; i++) begin
517         @(posedge clk);
518     end
519
520     $stop;
521 end
522
523 endmodule // DE1_SoC_testbench
524
525 /*****
526
527 fill_space.sv
528
529 *****/
530
531 /*
532  * Connor Aksama
533  * 01/30/2023
534  * CSE 371
535  * Lab 3
536 */
537
538 /**
539  * Cycles through each pixel on a screen with the size of the parameterized dimensions.
540  * Ex. (0,0) -> (1,0) -> (2,0) -> ... -> (638,479) -> (639, 479) -> (0,0)
541  * Outputs a new pixel on each clock cycle.
542
543  * Inputs:
544  *   clk [1 bit] - the clock to use for this module
545  *   reset [1 bit] - the reset signal for this module; resets to point (0,0)
546
547  * Outputs:
548  *   x [10 bit] - The x coordinate of the pixel to draw to
549  *   y [9 bit] - The y coordinate of the pixel to draw to
550 */
551 module fill_space #(
552     parameter width = 640, height = 480
553 )
554 (
555     output logic [9:0] x
556     ,output logic [8:0] y

```

```

557     ,input logic clk, reset
558 );
559
560 logic [9:0] x_d, x_q;
561 logic [8:0] y_d, y_q;
562
563 assign x = x_q;
564 assign y = y_q;
565
566 // Determine next coordinate
567 always_comb begin
568     if (x_q >= width - 1 && y_q >= height - 1) begin
569         // Reached x max and y max
570         x_d = 0;
571         y_d = 0;
572     end else if (x_q >= width - 1) begin
573         // Reached x max
574         x_d = 0;
575         y_d = y_q + 1;
576     end else begin
577         // Reached y max
578         x_d = x_q + 1;
579         y_d = y_q;
580     end
581 end
582
583 // Register outputs
584 always_ff @(posedge clk) begin
585     if (reset) begin
586         x_q <= 0;
587         y_q <= 0;
588     end else begin
589         x_q <= x_d;
590         y_q <= y_d;
591     end
592 end
593
594 endmodule // fill_space
595
596 // Module to test the functionality of the fill_space module
597 module fill_space_testbench();
598
599     logic [9:0] x;
600     logic [8:0] y;
601     logic clk, reset;
602
603     fill_space #(4,5) dut (.*);
604
605     parameter CLOCK_PERIOD = 100;
606     initial begin
607         clk <= 0;
608         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
609     end
610
611     initial begin
612         integer i;
613
614         @(posedge clk) reset <= 1;
615         @(posedge clk) reset <= 0;
616
617         // Run through each pixel in the screen
618         for (i = 0; i < 50; i++) begin
619             @(posedge clk);
620         end
621
622         $stop;
623     end
624
625 endmodule // fill_space_testbench

```

```

626
627 /*****
628
629 line_drawer.sv
630
631 *****/
632
633 /*
634  * Connor Aksama
635  * 01/30/2023
636  * CSE 371
637  * Lab 3
638 */
639
640 /**
641  * Given two endpoints (x0, y0), (x1, y1), outputs a sequence of points (x,y) that
642  * connect the two points.
643  * Outputs at most one new point (x,y) every clock cycle starting on the cycle after the
644  * input endpoints change.
645
646  * Inputs:
647  *   clk [1 bit] - the clock to use for this module
648  *   reset [1 bit] - the reset signal for this module; resets algorithm to first
649  *   iteration given inputs
650  *   x0 [10 bit] - The x coordinate of the first point of the line to draw
651  *   y0 [9 bit] - The y coordinate of the first point of the line to draw
652  *   x1 [10 bit] - The x coordinate of the second point of the line to draw
653  *   y1 [9 bit] - The y coordinate of the second point of the line to draw
654
655  * Outputs:
656  *   x [10 bit] - The x coordinate of the current pixel to draw
657  *   y [9 bit] - The y coordinate of the current pixel to draw
658 */
659 module line_drawer(
660     input logic clk, reset,
661
662     // x and y coordinates for the start and end points of the line
663     input logic [9:0] x0, x1,
664     input logic [8:0] y0, y1,
665
666     //outputs cooresponding to the coordinate pair (x, y)
667     output logic [9:0] x,
668     output logic [8:0] y
669 );
670
671 // Bresenham error
672 logic signed [11:0] error, p_error, n_error;
673
674 // Temp swap logic
675 logic signed [9:0] x_min, x_max, y_start, y_end;
676 logic signed [9:0] x0_r, x1_r, y0_r, y1_r;
677 // Bresenham temp calculations
678 logic is_steep, is_steep_r;
679 logic signed [10:0] delta_x;
680 logic signed [10:0] delta_y;
681 logic signed [9:0] step;
682
683 // Registered output
684 logic signed [9:0] px, nx;
685 logic signed [8:0] py, ny;
686 logic [9:0] prev_x0, prev_x1;
687 logic [8:0] prev_y0, prev_y1;
688
689 // Compute initial values before entering main loop
690 always_comb begin
691     delta_x = x1 - x0;
692     delta_y = y1 - y0;
693 
```

```

692     is_steep = (delta_y < 0 ? -delta_y : delta_y) > (delta_x < 0 ? -delta_x : delta_x
693 );
694
695     if (is_steep) begin
696         // Swap x, y
697         x0_r = y0;
698         y0_r = x0;
699         x1_r = y1;
700         y1_r = x1;
701     end else begin
702         x0_r = x0;
703         x1_r = x1;
704         y0_r = y0;
705         y1_r = y1;
706     end
707
708     if (x0_r > x1_r) begin
709         // swap p0, p1
710         x_min = x1_r;
711         x_max = x0_r;
712         y_start = y1_r;
713         y_end = y0_r;
714     end else begin
715         x_min = x0_r;
716         x_max = x1_r;
717         y_start = y0_r;
718         y_end = y1_r;
719     end
720
721     delta_x = x_max - x_min;
722     delta_y = (y_end - y_start < 0 ? -(y_end - y_start) : y_end - y_start);
723     error = -(delta_x / 2);
724
725     if (y_start < y_end) step = 1;
726     else                 step = -1;
727
728     if (p_error + delta_y >= 0 && px < x_max) begin
729         // Step x and y, compute next error
730         ny = py + step;
731         nx = px + 1;
732         n_error = p_error + delta_y - delta_x;
733     end else if (px < x_max) begin
734         // Step x, compute next error
735         ny = py;
736         nx = px + 1;
737         n_error = p_error + delta_y;
738     end else begin
739         // Reached end of line, keep outputting same point
740         nx = px;
741         ny = py;
742         n_error = p_error;
743     end
744
745     if (is_steep_r) begin
746         // Swap x,y
747         x = py;
748         y = px;
749     end else begin
750         x = px;
751         y = py;
752     end
753
754     end
755
756     // Update new values of current x,y in main loop
757     always_ff @(posedge clk) begin
758         prev_x0 <= x0;
759         prev_x1 <= x1;
760         prev_y0 <= y0;

```

```

760     prev_y1 <= y1;
761     is_steep_r <= is_steep;
762
763     if (reset || prev_x0 != x0 || prev_y0 != y0 || prev_x1 != x1 || prev_y1 != y1)
764         begin
765             // Reset initial values to input if input changes or reset
766             px <= x_min;
767             py <= y_start;
768             p_error <= error;
769         end else begin
770             // Output computed next values
771             px <= nx;
772             py <= ny;
773             p_error <= n_error;
774         end
775     end
776
777 endmodule // line_drawer
778
779 // Module to test the functionality of the line_drawer module
780 module line_drawer_testbench();
781
782     logic clk, reset;
783
784     logic [9:0] x0, x1;
785     logic [8:0] y0, y1;
786
787     logic [9:0] x;
788     logic [8:0] y;
789
790     line_drawer dut (.*) ;
791
792     parameter CLOCK_PERIOD = 100;
793     initial begin
794         clk <= 0;
795         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
796     end
797
798     initial begin
799         integer i;
800
801         @(posedge clk) reset <= 1; x0 <= 0; y0 <= 0; x1 <= 0; y1 <= 0;
802
803         // Horizontal
804         @(posedge clk) reset <= 0; x0 <= 0; y0 <= 0; x1 <= 10; y1 <= 0;
805         for (i = 0; i < 20; i++) begin
806             @(posedge clk);
807         end
808
809         // Vertical
810         @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 0; y1 <= 10;
811         for (i = 0; i < 20; i++) begin
812             @(posedge clk);
813         end
814
815         // Diagonal
816         @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 5; y1 <= 5;
817         for (i = 0; i < 20; i++) begin
818             @(posedge clk);
819         end
820
821         // Shallow
822         @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 10; y1 <= 2;
823         for (i = 0; i < 20; i++) begin
824             @(posedge clk);
825         end
826
827         // Steep

```

```

828         @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 2; y1 <= 10;
829         for (i = 0; i < 20; i++) begin
830             @(posedge clk);
831         end
832
833         $stop;
834     end
835
836 endmodule // line_drawer_testbench
837
838 /*****
839
840 VGA_framebuffer.sv
841
842 *****/
843
844 // VGA driver: provides I/O timing and double-buffering for the VGA port.
845
846 module VGA_framebuffer(
847     input logic clk, rst,
848     input logic [9:0] x, // The x coordinate to write to the buffer.
849     input logic [8:0] y, // The y coordinate to write to the buffer.
850     input logic pixel_color, pixel_write, // The data to write (color) and write-enable.
851
852     input logic dfb_en, // Double-Frame Buffer Enable
853
854     output logic frame_start, // Pulse is fired at the start of a frame.
855
856     // Outputs to the VGA port.
857     output logic [7:0] VGA_R, VGA_G, VGA_B,
858     output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N
859 );
860
861 /*
862 *
863 * HCOUNT 1599 0                1279                1599 0
864 *
865 * _____|_____ Video |_____|_____ Video
866 *
867 *
868 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
869 *
870 * |_____|_____ VGA_HS _____|_____|_____
871 *
872 */
873
874 // Constants for VGA timing.
875 localparam HPX = 11'd640*2, HFP = 11'd16*2, HSP = 11'd96*2, HBP = 11'd48*2;
876 localparam VLN = 11'd480, VFP = 10'd11, VSP = 10'd2, VBP = 10'd31;
877 localparam HTOTAL = HPX + HFP + HSP + HBP; // 800*2=1600
878 localparam VTOTAL = VLN + VFP + VSP + VBP; // 524
879
880 // Horizontal counter.
881 logic [10:0] h_count;
882 logic end_of_line;
883
884 assign end_of_line = h_count == HTOTAL - 1;
885
886 always_ff @(posedge clk)
887     if (rst) h_count <= 0;
888     else if (end_of_line) h_count <= 0;
889     else h_count <= h_count + 11'd1;
890
891 // Vertical counter & buffer swapping.
892 logic [9:0] v_count;
893 logic end_of_field;
894 logic front_odd; // whether odd address is the front buffer.
895
896 assign end_of_field = v_count == VTOTAL - 1;

```

```

897 assign frame_start = !h_count && !v_count;
898
899 always_ff @(posedge clk)
900     if (rst) begin
901         v_count <= 0;
902         front_odd <= 0;
903     end else if (end_of_line)
904         if (end_of_field) begin
905             v_count <= 0;
906             front_odd <= !front_odd;
907         end else
908             v_count <= v_count + 10'd1;
909
910 // Sync signals.
911 assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
912 assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
913 assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
914 assign VGA_SYNC_N = 1; // Unused by VGA
915
916 // Blank area signal.
917 logic blank;
918 assign blank = h_count >= HPX || v_count >= VLN;
919
920 // Double-buffering.
921 logic buffer[640*480*2-1:0];
922 logic [19:0] wr_addr, rd_addr;
923 logic rd_data;
924
925 assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
926 assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};
927
928 always_ff @(posedge clk) begin
929     if (pixel_write) buffer[wr_addr] <= pixel_color;
930     if (VGA_CLK) begin
931         rd_data <= buffer[rd_addr];
932         VGA_BLANK_N <= ~blank;
933     end
934 end
935
936 // Color output.
937 assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
938 endmodule
939

```