```
/**********************************

car_tracking.sv

**********************************/

/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * Controller module for the car tracking system.

 * Inputs:
 *      wr_num [16 bit] - The number of cars to track for the current hour
 *      wr_hour [3 bit] - The hour for which to update the number of cars
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the cyclic display to hour 0.

 * Outputs:
 *      curr_num [16 bit] - The number of cars that have entered in the corresponding hour
 *      curr_hour [3 bit] - The hour at which the corresponding number of cars have entered
*/
module car_tracking #(
        parameter clk_freq = 50000000, duration_sec = 1
    )(
        output logic [15:0] curr_num
        ,output logic [2:0] curr_hour
        ,input logic [15:0] wr_num
        ,input logic [2:0] wr_hour
        ,input logic clk, reset
    );

    localparam timer_target = clk_freq * duration_sec;

    logic [2:0] rd_addr, rd_out1;
    logic [31:0] timer;
    logic reset_timer;

    // RAM module for car tracking data
    // clk -> clock
    // wr_num -> data
    // rd_addr (current hour to read) -> rdaddress
    // wr_hour (current hour to modify) -> wraddress
    // wren -> always write
    // q -> curr_num (current num of cars)
    ram8x16 ram (
        .clock(clk)
        ,.data(wr_num)
        ,.rdaddress(rd_addr)
        ,.wraddress(wr_hour)
        ,.wren('1)
        ,.q(curr_num)
    );

    always_comb begin
        // Reset timer at target
        reset_timer = (timer >= timer_target - 1);
    end

    always_ff @(posedge clk) begin
        if (reset) begin
            // Reset cycle to beginning
            rd_addr <= '0;
```

```
            rd_out1 <= '0;
            timer <= '0;
        end else if (reset_timer) begin
            // Reset timer, move to next hour
            timer <= '0;
            rd_addr <= rd_addr + 3'd1;
        end else begin
            // Increment timer
            timer <= timer + 32'd1;
        end

        // Sync with RAM output
        curr_hour <= rd_addr;
    end

endmodule  // car_tracking

/*
 * Testbench to test the functionality of the car_tracking module
 */
`timescale 1 ps / 1 ps
module car_tracking_testbench();

    logic [15:0] curr_num;
    logic [2:0] curr_hour;

    logic [15:0] wr_num;
    logic [2:0] wr_hour;
    logic clk, reset;

    car_tracking #(.clk_freq(2), .duration_sec(1)) dut (.*);

        parameter CLOCK_PERIOD = 100;
    initial begin
        clk <= '0;
        forever #(CLOCK_PERIOD / 2) clk <= ~clk;
    end

    initial begin
        integer i;

        @(posedge clk) reset <= '1; wr_num <= '0; wr_hour <= '0;
        @(posedge clk) reset <= '0;

        // Write data to RAM
        for (i = 0; i < 10; i++) begin
            @(posedge clk) wr_hour <= i;
            if (i % 2 == 0) begin
                wr_num <= wr_num + 15'd1;
            end
        end

        // Cycle through values
        for (i = 0; i < 30; i++) begin
            @(posedge clk);
        end


        $stop;
    end

endmodule  // car_tracking_testbench

/**********************************

DE1_SoC.sv

**********************************/

```

```
/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * Top-level module for Lab 6 Task 2. Instantiates rush hour and car tracking modules and defines logic to connect I/O to peripherals.

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment display format.
 *  HEX2 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment display format.
 *  HEX3 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment display format.

 * Inouts:
 *  V_GPIO [13 bit] - Virtual GPIO Ports
*/
module DE1_SoC #(
        parameter clk_freq = 50000000, display_duration_sec = 1
    )(
        output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
        ,input logic [9:0] SW
        ,input logic [3:0] KEY
        ,input logic CLOCK_50
        ,inout logic [35:23] V_GPIO
    );

    logic clk;
    assign clk = CLOCK_50;

    // Which parking spaces occupied?
    logic [2:0] pp;
    assign pp = V_GPIO[30:28];

    // Car waiting at gate
    logic entr_wait, exit_wait;
    assign entr_wait = V_GPIO[23];
    assign exit_wait = V_GPIO[24];

    // Light LEDs at each parking spot depending on presence
    assign {V_GPIO[32], V_GPIO[27], V_GPIO[26]} = pp;

    // Full LED
    logic led_full;
    assign V_GPIO[34] = led_full;

    // Ctrl for gates
    logic entr_open, exit_open;
    assign V_GPIO[31] = entr_open;
    assign V_GPIO[33] = exit_open;

    // Remaining spot HEX generator
    logic [1:0] rem_num;
```

```
199         logic [6:0] rem_hex;
200         seg7 rem_spot (.HEX(rem_hex), .num({2'b0, rem_num}));
201 
202         // Current hour HEX generator
203         logic [2:0] curr_hour;
204         logic [6:0] ch_hex;
205         seg7 hour (.HEX(ch_hex), .num({1'b0, curr_hour}));
206 
207         logic reset;
208         assign reset = SW[9];
209 
210         logic done;
211 
212         // Rush Hour Tracker
213         // rh_start -> rh_start (saved rush hour start)
214         // rh_end -> rh_end (saved rush hour end)
215         // rh_start_valid -> rh_start_valid (1 if rh_start data valid)
216         // rh_end_valid -> rh_end_valid (1 if rh_end data valid)
217         // pp -> pp
218         // curr_hour -> hour (current hour)
219         // clk -> clk
220         // reset -> reset
221         logic [2:0] rh_start, rh_end;
222         logic rh_start_valid, rh_end_valid;
223         rh_ctrl rush_hour (
224             .rh_start(rh_start)
225             ,.rh_end(rh_end)
226             ,.rh_start_valid(rh_start_valid)
227             ,.rh_end_valid(rh_end_valid)
228             ,.pp(pp)
229             ,.hour(curr_hour)
230             ,.clk(clk)
231             ,.reset(reset)
232         );
233 
234         logic [6:0] end_hex, start_hex;
235         // Rush Hour End HEX generator
236         seg7 end_hour (.HEX(end_hex), .num({1'b0, rh_end}));
237         // Rush Hour Start HEX generator
238         seg7 start_hour (.HEX(start_hex), .num({1'b0, rh_start}));
239 
240         // Num Car Tracker/Display
241         // curr_num -> num_cars_display (Output for saved # of cars for curr_hour_display)
242         // curr_hour -> curr_hour_display (Current hour for which to display # of cars)
243         // total_cars -> wr_num (write the total number of cars entered so far)
244         // curr_hour -> wr_hour (current hour, write total num cars for this hour)
245         // clk -> clk
246         // reset -> reset
247         logic [15:0] total_cars, num_cars_display;
248         logic [2:0] curr_hour_display;
249         car_tracking #(
250             .clk_freq(clk_freq), .duration_sec(display_duration_sec)
251         ) results (
252             .curr_num(num_cars_display)
253             ,.curr_hour(curr_hour_display)
254             ,.wr_num(total_cars)
255             ,.wr_hour(curr_hour)
256             ,.clk(clk)
257             ,.reset(reset)
258         );
259 
260         logic [6:0] ncd_hex, chd_hex;
261         // EOD num cars HEX generator
262         seg7 ncd (.HEX(ncd_hex), .num(num_cars_display[3:0]));
263         // EOD curr hour HEX generator
264         seg7 chd (.HEX(chd_hex), .num({1'b0, curr_hour_display}));
265 
266         logic inc_hour;
267         // Generate a one cycle pulse for the inc_hour signal (KEY[0])
```

```
268         pulse_gen gen_hour (.pulse(inc_hour), .in(~KEY[0]), .clk(clk), .reset(reset));
269
270         logic inc_cars;
271         pulse_gen gen_cars (.pulse(inc_cars), .in(entr_open), .clk(clk), .reset(reset));
272
273         always_comb begin
274             // Ctrl signals
275             led_full = (pp == 3'b111);
276             entr_open = entr_wait & (pp != 3'b111);
277             exit_open = exit_wait;
278
279             // Find remaining number of spots
280             case (pp)
281                 3'b000:
282                     rem_num = 2'd3;
283                 3'b001, 3'b010, 3'b100:
284                     rem_num = 2'd2;
285                 3'b011, 3'b101, 3'b110:
286                     rem_num = 2'd1;
287                 3'b111:
288                     rem_num = 2'd0;
289             endcase
290
291             // ctrl driver for HEX outputs
292             if (done) begin
293                 HEX5 = ~7'b0;
294                 if (rh_end_valid) begin
295                     HEX4 = end_hex;
296                 end else begin
297                     HEX4 = ~7'b1000000;
298                 end
299                 if (rh_start_valid) begin
300                     HEX3 = start_hex;
301                 end else begin
302                     HEX3 = ~7'b1000000;
303                 end
304                 HEX2 = chd_hex;
305                 HEX1 = ncd_hex;
306                 HEX0 = ~7'b0;
307             end else if (led_full) begin
308                 HEX5 = ch_hex;
309                 HEX4 = ~7'b0;
310                 HEX3 = ~7'b1110001;  // F
311                 HEX2 = ~7'b0111110;  // U
312                 HEX1 = ~7'b0111000;  // L
313                 HEX0 = ~7'b0111000;  // L
314             end else begin
315                 HEX5 = ch_hex;
316                 HEX4 = ~7'b0;
317                 HEX3 = ~7'b0;
318                 HEX2 = ~7'b0;
319                 HEX1 = ~7'b0;
320                 HEX0 = rem_hex;
321             end
322         end
323
324         always_ff @(posedge clk) begin
325             if (reset) begin
326                 // Reset work day
327                 curr_hour <= '0;
328                 done <= '0;
329             end else if (inc_hour && curr_hour == 3'd7) begin
330                 // EOD reached
331                 done <= '1;
332             end else if (inc_hour) begin
333                 // Increase work hour
334                 curr_hour <= curr_hour + 3'b001;
335             end
336
```

```
337             if (reset) begin
338                 // Reset # total cars
339                 total_cars <= '0;
340             end else if (inc_cars) begin
341                 // Car entering
342                 total_cars <= total_cars + 16'b1;
343             end
344         end
345 
346     endmodule  // DE1_SoC
347 
348     /*
349      * Testbench to test the functionality of the DE1_SoC module
350      */
351     `timescale 1 ps / 1 ps
352     module DE1_SoC_testbench();
353 
354         logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
355         logic [9:0] SW;
356         logic [3:0] KEY;
357         logic CLOCK_50;
358         wire [35:23] V_GPIO;
359 
360         logic clk;
361         assign CLOCK_50 = clk;
362 
363         logic reset;
364         assign SW[9] = reset;
365 
366         logic inc_hour;
367         assign KEY[0] = ~inc_hour;
368 
369         logic [2:0] pp;
370         assign V_GPIO[30:28] = pp;
371 
372         logic entr_wait, exit_wait;
373         assign V_GPIO[23] = entr_wait;
374         assign V_GPIO[24] = exit_wait;
375 
376         logic [2:0] led_p;
377         assign led_p = {V_GPIO[32], V_GPIO[27], V_GPIO[26]};
378 
379         logic led_full;
380         assign led_full = V_GPIO[34];
381 
382         logic entr_open, exit_open;
383         assign entr_open = V_GPIO[31];
384         assign exit_open = V_GPIO[33];
385 
386         DE1_SoC #(.clk_freq(3), .display_duration_sec(1)) dut (.*);
387 
388         parameter CLOCK_PERIOD = 100;
389         initial begin
390             clk <= '0;
391             forever #(CLOCK_PERIOD / 2) clk <= ~clk;
392         end
393 
394         initial begin
395             integer i, j;
396 
397             @(posedge clk) reset <= '1; inc_hour <= '0; pp <= '0; entr_wait <= '0; exit_wait
                <= '0;
398             @(posedge clk) reset <= '0;
399 
400             for (i = 0; i < 8; i++) begin
401                 if (i % 2 == 0) begin
402                     // Three cars enter
403                     for (j = 0; j < 3; j++) begin
404                         @(posedge clk) entr_wait <= '1;
```

```
405                     @(posedge clk);
406                     if (entr_open) begin
407                         entr_wait <= '0;
408                         pp[j] <= '1;
409                     end else begin
410                         j--;
411                     end
412                 end
413             end else begin
414                 // Three cars leave
415                 for (j = 0; j < 3; j++) begin
416                     @(posedge clk) exit_wait <= '1;
417                     @(posedge clk);
418                     if (exit_open) begin
419                         exit_wait <= '0;
420                         pp[j] <= '0;
421                     end else begin
422                         j--;
423                     end
424                 end
425             end
426             @(posedge clk) inc_hour <= '1;
427             @(posedge clk) inc_hour <= '0;
428         end
429 
430         for (i = 0; i < 32; i++) begin
431             @(posedge clk);
432         end
433 
434         @(posedge clk) reset <= '1; inc_hour <= '0; pp <= '0; entr_wait <= '0; exit_wait
        <= '0;
435         @(posedge clk) reset <= '0;
436         for (i = 0; i < 3; i++) begin
437             @(posedge clk) entr_wait <= '1;
438             @(posedge clk);
439             if (entr_open) begin
440                 entr_wait <= '0;
441                 pp[j] <= '1;
442             end
443         end
444         for (i = 0; i < 8; i++) begin
445             @(posedge clk) inc_hour <= '1;
446             @(posedge clk) inc_hour <= '0;
447         end
448         for (i = 0; i < 32; i++) begin
449             @(posedge clk);
450         end
451         $stop;
452     end
453 
454 endmodule  // DE1_SoC_testbench
455 
456 /**********************************
457 
458 pulse_gen.sv
459 
460 **********************************/
461 
462 /*
463  * Connor Aksama
464  * 03/13/2023
465  * CSE 371
466  * Lab 6
467 */
468 
469 /**
470  * One cycle pulse generator for arbitrary length input.
471 
472  * Inputs:
```

```
 *      in [1 bit] - Input for which to generate pulse
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the cyclic display to hour 0.

 * Outputs:
 *      pulse [1 bit] - The output signal where pulse is sent
*/
module pulse_gen (
        output logic pulse
        ,input logic in, clk, reset
    );

    typedef enum logic [1:0] { s_wait_rise, s_pulse, s_wait_fall } state;

    state ps, ns;
    logic in_reg;

    always_comb begin
        ns = ps;

        // Handle state transitions
        case (ps)
            s_wait_rise: begin
                if (in_reg) begin
                    ns = s_pulse;
                end
            end
            s_pulse: begin
                if (~in_reg) begin
                    ns = s_wait_rise;
                end else begin
                    ns = s_wait_fall;
                end
            end
            s_wait_fall: begin
                if (~in_reg) begin
                    ns = s_wait_rise;
                end
            end
        endcase

        pulse = (ps == s_pulse);
    end

    always_ff @(posedge clk) begin
        // Update state
        if (reset) begin
            ps <= s_wait_rise;
        end else begin
            ps <= ns;
        end
        // Register input
        in_reg <= in;
    end

endmodule  // pulse_gen

/*
 * Testbench to test the functionality of the pulse_gen module
 */
module pulse_gen_testbench();

    logic pulse, in, clk, reset;

    pulse_gen dut (.*);

    parameter CLOCK_PERIOD = 100;
    initial begin
        clk <= '0;
```

```
542             forever #(CLOCK_PERIOD / 2) clk <= ~clk;
543         end
544 
545         initial begin
546             integer i;
547 
548             @(posedge clk) reset <= '1; in <= '0;
549             @(posedge clk) reset <= '0; in <= '1;
550 
551             for (i = 0; i < 5; i++) begin
552                 @(posedge clk);
553             end
554 
555             @(posedge clk) in <= '0;
556 
557             for (i = 0; i < 5; i++) begin
558                 @(posedge clk);
559             end
560 
561             @(posedge clk) in <= '1;
562             @(posedge clk) in <= '0;
563 
564             for (i = 0; i < 5; i++) begin
565                 @(posedge clk);
566             end
567 
568             $stop;
569         end
570 
571     endmodule  // pulse_gen_testbench
572 
573     /**********************************
574 
575     rh_ctrl.sv
576 
577     **********************************/
578 
579     /*
580      * Connor Aksama
581      * 03/13/2023
582      * CSE 371
583      * Lab 6
584     */
585 
586     /**
587      * Controller module for the rush hour system.
588 
589      * Inputs:
590      *      pp [3 bit] - Parking spaces currently occupied
591      *      hour [3 bit] - The current hour
592      *      clk [1 bit] - The clock to use for this module.
593      *      reset [1 bit] - Resets the FSM to its initial state.
594 
595      * Outputs:
596      *      rh_start [3 bit] - The saved start of rush hour
597      *      rh_end [3 bit] - The saved end of rush hour
598      *      rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
599      *      rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
600     */
601     module rh_ctrl (
602             output logic [2:0] rh_start, rh_end
603             ,output logic rh_start_valid, rh_end_valid
604             ,input logic [2:0] pp, hour
605             ,input logic clk, reset
606         );
607 
608 
609         typedef enum { s_normal, s_rush, s_end } state;
610 
```

```
611       state ps, ns;
612
613       logic full, empty, hour0;
614       logic save_start, save_end, reset_data;
615
616       // Datapath module
617       // rh_start -> rh_start (saved start hour)
618       // rh_end -> rh_end (saved end hour)
619       // rh_start_valid -> rh_start_valid (rh_start valid?)
620       // rh_end_valid -> rh_end_valid (rh_end valid?)
621       // hour -> hour (current hour of system)
622       // save_start -> save_start (current hour is start)
623       // save_end -> save_end (current hour is end)
624       // clk -> clk
625       // reset | reset_data -> reset (reset FSM to start)
626       rh_data rush_hour_data (
627           .rh_start(rh_start)
628           ,.rh_end(rh_end)
629           ,.rh_start_valid(rh_start_valid)
630           ,.rh_end_valid(rh_end_valid)
631           ,.hour(hour)
632           ,.save_start(save_start)
633           ,.save_end(save_end)
634           ,.clk(clk)
635           ,.reset(reset | reset_data)
636       );
637
638       always_comb begin
639
640           // Handle state transitions
641           case (ps)
642
643               s_normal: begin
644                   if (full) begin
645                       ns = s_rush;
646                   end else begin
647                       ns = s_normal;
648                   end
649               end
650
651               s_rush: begin
652                   if (empty) begin
653                       ns = s_end;
654                   end else begin
655                       ns = s_rush;
656                   end
657               end
658
659               s_end: begin
660                   if (hour0) begin
661                       ns = s_normal;
662                   end else begin
663                       ns = s_end;
664                   end
665               end
666
667           endcase
668
669           // Control signals
670           save_start = (ps == s_normal) & full;
671           save_end = (ps == s_rush) & empty;
672           reset_data = (ps == s_end) & hour0;
673
674       end
675
676
677       always_ff @(posedge clk) begin
678           // Update FSM
679           if (reset) begin
```

```
            ps <= s_normal;
        end else begin
            ps <= ns;
        end

        // Register control signals
        full <= (pp == 3'b111);
        empty <= (pp == 3'b000);
        hour0 <= (hour == 3'b000);
    end


endmodule  // rh_ctrl

/*
 * Testbench to test the functionality of the rh_ctrl module
 */
module rh_ctrl_testbench();

    logic [2:0] rh_start, rh_end;
    logic rh_start_valid, rh_end_valid;

    logic [2:0] pp, hour;
    logic clk, reset;

    rh_ctrl dut (.*);

    parameter CLOCK_PERIOD = 100;
    initial begin
        clk <= '0;
        forever #(CLOCK_PERIOD / 2) clk <= ~clk;
    end

    initial begin
        integer i;

        @(posedge clk) reset <= '1; pp <= '0; hour <= '0;
        @(posedge clk) reset <= '0;

        for (i = 0; i < 40; i++) begin
            @(posedge clk);
            @(posedge clk)
            if (i % 2 == 0) begin
                // Increment hour
                hour <= hour + 3'd1;
            end
            if (i % 3 == 0) begin
                // Add car to lot
                pp <= pp + 3'd1;
            end
        end

        $stop;
    end

endmodule  // rh_ctrl_testbench

/**********************************

rh_data.sv

**********************************/

/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/
```

```
749
750  /**
751   * Datapath module for the rush hour system.
752
753   * Inputs:
754   *      hour [3 bit] - The current hour
755   *      save_start [1 bit] - 1 if hour should be saved for start, 0 o.w.
756   *      save_end [1 bit] - 1 if hour should be saved for end, 0 o.w.
757   *      clk [1 bit] - The clock to use for this module.
758   *      reset [1 bit] - Resets the FSM to its initial state.
759
760   * Outputs:
761   *      rh_start [3 bit] - The saved start of rush hour
762   *      rh_end [3 bit] - The saved end of rush hour
763   *      rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
764   *      rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
765  */
766  module rh_data (
767          output logic [2:0] rh_start, rh_end
768          ,output logic rh_start_valid, rh_end_valid
769          ,input logic [2:0] hour
770          ,input logic save_start, save_end, clk, reset
771      );
772
773
774      always_ff @(posedge clk) begin
775          // Register hour/valid bits based on ctrl signals
776          if (reset) begin
777              rh_start <= '0;
778              rh_end <= '0;
779              rh_start_valid <= '0;
780              rh_end_valid <= '0;
781          end else if (save_start) begin
782              rh_start <= hour;
783              rh_start_valid <= '1;
784          end else if (save_end) begin
785              rh_end <= hour;
786              rh_end_valid <= '1;
787          end
788
789
790      end
791
792  endmodule  // rh_data
793
794  /*
795   * Testbench to test the functionality of the rh_ctrl module
796   */
797  module rh_data_testbench();
798
799      logic [2:0] rh_start, rh_end;
800      logic rh_start_valid, rh_end_valid;
801      logic [2:0] hour;
802      logic save_start, save_end, clk, reset;
803
804      rh_data dut (.*);
805
806      parameter CLOCK_PERIOD = 100;
807      initial begin
808          clk <= '0;
809          forever #(CLOCK_PERIOD / 2) clk <= ~clk;
810      end
811
812      initial begin
813          integer i;
814
815          @(posedge clk) reset <= '1; save_start <= '0; save_end <= '0; hour <= '0;
816          @(posedge clk) reset <= '0;
817
```

```
818             // Do nothing
819             for (i = 0; i < 10; i++) begin
820                 @(posedge clk);
821                 hour <= i;
822             end
823 
824             // Save start hour
825             @(posedge clk) save_start <= '1;
826             @(posedge clk) save_start <= '0;
827 
828             for (i = 0; i < 10; i++) begin
829                 @(posedge clk);
830             end
831 
832             // Save end hour
833             @(posedge clk); hour <= hour + 3;
834             @(posedge clk); save_end <= '1;
835             @(posedge clk); save_end <= '0;
836 
837             for (i = 0; i < 10; i++) begin
838                 @(posedge clk);
839             end
840 
841             @(posedge clk); hour <= '0; reset <= '1;
842             @(posedge clk); reset <= '0;
843 
844             for (i = 0; i < 10; i++) begin
845                 @(posedge clk);
846             end
847 
848             $stop;
849         end
850 
851     endmodule  // rh_data_testbench
852 
853     /*********************************
854 
855     seg7.sv
856 
857     *********************************/
858 
859     /*
860      * Connor Aksama
861      * 03/13/2023
862      * CSE 371
863      * Lab 6
864     */
865 
866     /**
867      * Defines data for 1-digit hexadecimal HEX display given a 4-bit unsigned integer
868 
869      * Inputs:
870      *       num [4 bit] - An unsigned integer value [0x0-0xF] to display
871 
872      * Outputs:
873      *       HEX [7 bit] - A HEX display for the input num, formatted in standard seven
       segment display format
874     */
875     module seg7(
876         output logic [6:0] HEX
877         ,input logic [3:0] num
878         );
879 
880         // Drive HEX output signals given num
881         always_comb begin
882             // Light HEX using num
883             case (num)
884                 //     Light: 6543210
885                  0: HEX = ~7'b0111111;  // 0
```

```
                1: HEX = ~7'b0000110;  // 1
                2: HEX = ~7'b1011011;  // 2
                3: HEX = ~7'b1001111;  // 3
                4: HEX = ~7'b1100110;  // 4
                5: HEX = ~7'b1101101;  // 5
                6: HEX = ~7'b1111101;  // 6
                7: HEX = ~7'b0000111;  // 7
                8: HEX = ~7'b1111111;  // 8
                9: HEX = ~7'b1101111;  // 9
               10: HEX = ~7'b1110111;  // A
               11: HEX = ~7'b1111100;  // b
               12: HEX = ~7'b1011000;  // c
               13: HEX = ~7'b1011110;  // d
               14: HEX = ~7'b1111001;  // E
               15: HEX = ~7'b1110001;  // F
              default: HEX = 7'bX;
            endcase
        end

    endmodule  // seg7

    /*
     * Tests the functionality of the seg7 module.
    */
    module seg7_testbench();

        logic [6:0] HEX;
        logic [3:0] num;

        seg7 dut (.HEX, .num);

        initial begin

            integer i;

            // Check HEX displays for integers 0x0-0xff
            for (i = 0; i <= 15; i++) begin
                #10 num = i;
            end
            #50;
        end

    endmodule  // seg7_testbench

```