E2.1 – Digital Electronics II

Solution Sheet 2 – Verilog HDL

1A. The question may be a bit ambiguous. Essentially this is similar to a HEX to 7 segment decoder, except that the 4-bit input is expected to be between 1 to 12 (hours), and the output should drive TWO 7 segment displays showing the tens and units (instead of a hexadecimal number). You can easily modify the decoder in the notes using case statement and concatenation construct. {HEX1_D, HEX0_D} concatenates two 7 bit variables to form a single 14-bit variable.

```
module hex_to_BCD (HEX1_D, HEX0_D, hex_num);

    output  [6:0] HEX1_D;      // the tenth digit
    output  [6:0] HEX0_D;      // the unit digit
  input [3:0] hex_num;    // 4-bit binary containing a

  reg   [6:0] HEX1_D;
  reg   [6:0] HEX0_D;

  always @ (hex_num)
    case (hex_num)
     4'h0: {HEX1_D, HEX0_D} = {7'b1111111,7'b1000000};
     4'h1: {HEX1_D, HEX0_D} = {7'b1111111,7'b1111001};
     4'h2: {HEX1_D, HEX0_D} = {7'b1111111,7'b0100100};
     4'h3: {HEX1_D, HEX0_D} = {7'b1111111,7'b0110000};
     4'h4: {HEX1_D, HEX0_D} = {7'b1111111,7'b0011001};
     4'h5: {HEX1_D, HEX0_D} = {7'b1111111,7'b0010010};
     4'h6: {HEX1_D, HEX0_D} = {7'b1111111,7'b0000010};
     4'h7: {HEX1_D, HEX0_D} = {7'b1111111,7'b1111000};
     4'h8: {HEX1_D, HEX0_D} = {7'b1111111,7'b0000000};
     4'h9: {HEX1_D, HEX0_D} = {7'b1111111,7'b0011000};
     4'ha: {HEX1_D, HEX0_D} = {7'b1111001,7'b1000000};
     4'hb: {HEX1_D, HEX0_D} = {7'b1111001,7'b1111001};
     4'hc: {HEX1_D, HEX0_D} = {7'b1111001,7'b0100100};
     4'hd: {HEX1_D, HEX0_D} = {7'b1111001,7'b0110000};
     4'he: {HEX1_D, HEX0_D} = {7'b1111001,7'b0011001};
     4'hf: {HEX1_D, HEX0_D} = {7'b1111001,7'b0010010};
    endcase
endmodule
```

2A. To generate a 1 Hz square signal from a 50MHz clock, you should divide the 50MHz by 25,000,000 to give a 2Hz clock with a counter, and then divide this 2Hz clock using a toggle flipflop.

The ÷N operation can be expressed in Verilog as either a up counter from zero to N-1, or a down counter from N-1 to zero. Using an up counter is usually less desirable because it N is user changeable (and NOT a constant), you need a binary comparator. With a down counter, you always detect zero no matter whether N is a constant or not.

```
module div_by_50million (tick,clk50);

    output          tick;       // a 1 second clock tick
    input           clk50;      // DE0's fixed 50MHz system clock signal

    reg     [24:0]  count;      // to count 25 million, need 25 bit counter
    reg     tick;

    parameter TC = 25'd25000000-25'd1;
    initial
      begin
        count = 0;
        tick = 0;
      end

    always @ (posedge clk50)
      begin
        if (count==0) begin
            count <= TC;
            tick <= ~tick;
          end   // ... if
        else
            count <= count-1'b1;
      end     //... always

endmodule
```

Here the time constant value of 24,999,999 is specified using the **parameter** keyword as a symbolic constant. This is good practice – you can change the period of the clock by changing this value without having to dig into the code and find out where the time constant is used. The `tick <= ~tick;` statement will produce a toggle fiipflop. The assignment <= will ensure that tick is only changed at the end of the always block, and therefore the synthesis system will generate a flipflop to store the previous value, only updating on positive edge of the clock.

To test this module, I used this testbench as the top-level module: it uses the 1 second clock to drive the decimal point (DP) of the HEX display 2. This will produce a blinking dot with a 1 second period.

```
module second_counter   (HEX2_DP, CLK50);

    output  HEX2_DP;    // HEX2 display decimal point
    input   CLK50;      // DE0's fixed 50MHz system clock signal

    div_by_50million BIGDIVIDER (HEX2_DP, CLK50);

endmodule
```

3A. This exercise shows how you may implement a shift register, initialise it to a one-hot pattern, and then rotate the pattern using the 1sec clock. The code is actually very simple:

```
module shift_pattern (pattern, clock);

    output  [9:0]   pattern;    // pattern to be displayed on DE0 LEDs
    input           clock;      // each clock tick is one second

    reg     [9:0]   pattern;    // pattern on green LEDs

    initial
        pattern <= 10'b1;

    always @ (posedge clock)
      begin
            pattern[1] <= pattern[0];
            pattern[2] <= pattern[1];
            pattern[3] <= pattern[2];
            pattern[4] <= pattern[3];
            pattern[5] <= pattern[4];
            pattern[6] <= pattern[5];
            pattern[7] <= pattern[6];
            pattern[8] <= pattern[7];
            pattern[9] <= pattern[8];
            pattern[0] <= pattern[9];
      end           // ... always
endmodule
```

4B. This differs from Q3 in that you need to remember the direction (using an extra FF). All you need to do is to introduce a new reg variable direction, and toggles this each time the pattern 1 reaches the end of the left or right. The first part of the solution is simple, a simple modification on Q3 to either shift right or left (not rotating) depending on the state of direction (opposite).

The detection of when to change direction is a bit more tricky. Remember that you must detect that it is about to reach the left-most or right-most LEDs on the cycle before. Here is my solution:

```
always @ (posedge clock)        // test for change of direction
  begin
    case ({direction, pattern[8], pattern[1]})
      3'b010:   direction <= ~direction;  // left shifting LED8 lit, change direction next time
      3'b101:   direction <= ~direction;  // right shifting LED1 lit, change direction next time
      default:  direction <= direction;   // ... otherwise, stay in the same direction
    endcase
  end
```

```
module shift_pattern (pattern, clock);

    output  [9:0]   pattern;    // pattern to be displayed on DE0 LEDs
    input           clock;      // each clock tick is one second

    reg     [9:0]   pattern;    // pattern on green LEDs
    reg             direction;  // pattern goes right (1) or left (0)

    initial
      begin
        pattern <= 10'b1;
        direction  <= 1'b0;
      end

    always @ (posedge clock)
      begin
        if (direction==0)
          begin     // shift left
            pattern[1] <= pattern[0];
            pattern[2] <= pattern[1];
            pattern[3] <= pattern[2];
            pattern[4] <= pattern[3];
            pattern[5] <= pattern[4];
            pattern[6] <= pattern[5];
            pattern[7] <= pattern[6];
            pattern[8] <= pattern[7];
            pattern[9] <= pattern[8];
            pattern[0] <= 1'b0;
          end       // ... if
        else
          begin     // shift right
            pattern[0] <= pattern[1];
            pattern[1] <= pattern[2];
            pattern[2] <= pattern[3];
            pattern[3] <= pattern[4];
            pattern[4] <= pattern[5];
            pattern[5] <= pattern[6];
            pattern[6] <= pattern[7];
            pattern[7] <= pattern[8];
            pattern[8] <= pattern[9];
            pattern[9] <= 1'b0;
          end       //... else
      end           // ... always

.....

endmodule
```

5B. This is fairly straightforward:

```
module hour_counter (hours, minute_ticks);

    output  [3:0]   hours;          // 4-bit count of hours
    input   minute_ticks;           // each tick is one minute

    reg     [3:0]   hours;

    initial
        hours <= 0;

    always @ (posedge minute_ticks)
        begin
          hours <= hours + 1'b1;
          if (hours == 12) hours <= 1;  // wrap around
        end
endmodule
```

It is now worthwhile for you to combine Q1-Q5 together and produce a testbench which displays a count for the second ticks on the lower two hex digits, a blinking DP for each second, and the hour count on the two two hex digits, while the green LEDs showing a pattern. Here is the top-level testbench file (called 'test'):

```
module test (HEX3_D, HEX2_D, HEX2_DP,
             HEX1_D, HEX0_D, LEDG,
             CLOCK_50);

    output  [6:0]   HEX3_D;     // 7 segment highest digit
    output  [6:0]   HEX2_D;     // 7 segment middle digit
    output  [6:0]   HEX1_D;     // 7 segment middle digit
    output  [6:0]   HEX0_D;     // 7 segment lowest digit
    output          HEX2_DP;    // Decimal point of HEX 2 display
    output  [9:0]   LEDG;       // 10 green LEDs
    input           CLOCK_50;   // DE0's 50MHz clock signal

    wire    [7:0]   count;      // count seconds
    wire    [3:0]   hours;      // count hours
    wire            ticks;      // link wire for second tick clock

// test for q2
    clock_divider   DIV     (ticks, CLOCK_50);
    tick_counter    TCNT    (count, ticks);
    hex_to_7seg     HEX0    (HEX0_D, count[3:0]);
    hex_to_7seg     HEX1    (HEX1_D, count[7:4]);
    assign  HEX2_DP = ticks;

// test for q4
    shift_pattern   SR      (LEDG, ticks);

// test for q5
    hour_counter    HR_CNT  (hours, ticks);   // use second_clock f
    hex_to_BCD      DECODER (HEX3_D, HEX2_D, hours);

endmodule
```

The module tick_counter code is simple:

```
module tick_counter (count, ticks);

    output  [7:0]   count;  // 8 bit count value to count elasped seconds
    input   ticks;          // each tick is one second

    reg     [7:0]   count;

    initial
      count <= 0;

    always @ (posedge ticks)
        count <= count + 1'b1;  // increment second counter

endmodule
```

6B. Here is the solution for a generic 60 ticks counter. It can be used to count minutes or to count seconds:

```
module counter_60   (tens, units, timeout, clock_tick);

    output  [3:0]   tens;           // BCD digital for tens
    output  [3:0]   units;          // BCD digital for units
    output          timeout;        // goes high for 1 tick period after 60 ticks
    input           clock_tick;

    reg     [3:0]   tens;
    reg     [3:0]   units;
    reg             timeout;

    initial
      begin
        tens <= 0;
        units <= 0;
      end

    always @ (posedge clock_tick)
      begin
        timeout <= 0;               // reset timeout signal
        if (units != 9)
            units <= units + 1'b1;
          else begin
            units <= 0;
            if (tens != 5)
              tens <= tens + 1'b1;
            else
              begin
                tens <= 0;
                timeout <= 1'b1;
              end
          end
      end
endmodule
```

7C. You should be able to combine everything together to produce a working digital clock in hardware. Initialising the clock to the correct time is bit trickier. My solution is to use SW0 and SW1 to set minutes and hours respectively. When the switch is OFF, the clock works normally. When it is ON, then a 1 second clock is fed to either the hour or minute counter. Therefore setting the correct time is just a matter of waiting until the correct minute or hour is reached and then return the switch to the OFF position.

You can download my complete solution as a zip file from my course webpage.