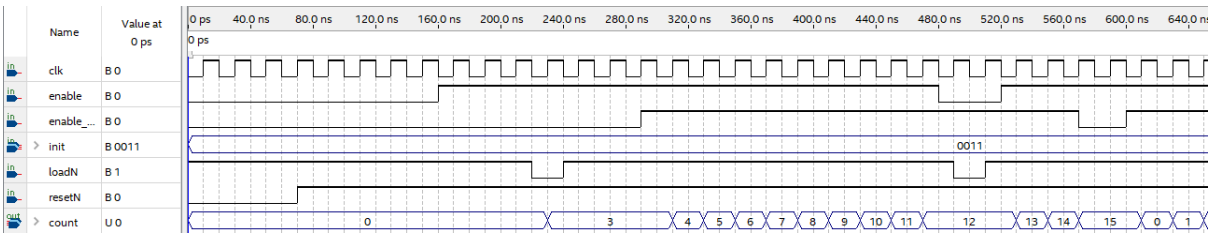
counter code:

```
1  // loadN, enable_cnt and enable to control the count
2  // and data input - init[3:0] for the load functionality
3
4  module up_counter
5     (
6     // Input, Output Ports
7     input logic clk,
8     input logic resetN,
9     input logic enable,
10    input logic loadN,
11    input logic enable_cnt,
12    input logic [3:0] init,
13    output logic [3:0] count
14    );
15
16 //-------------------------------------------------------
17    always_ff @( posedge clk or negedge resetN )
18    begin
19
20       if ( !resetN ) begin // Asynchronic reset
21        count <= 4'b0000;
22        end
23       else if (enable) begin
24          if (!loadN)
25             count <= init;
26          else if (enable_cnt)
27             count <= count + 4'b0001;
28       end
29    else count <= count;
30
31    end // always
32 //-------------------------------------------------------
33 endmodule
```

Verification with cross inputs:



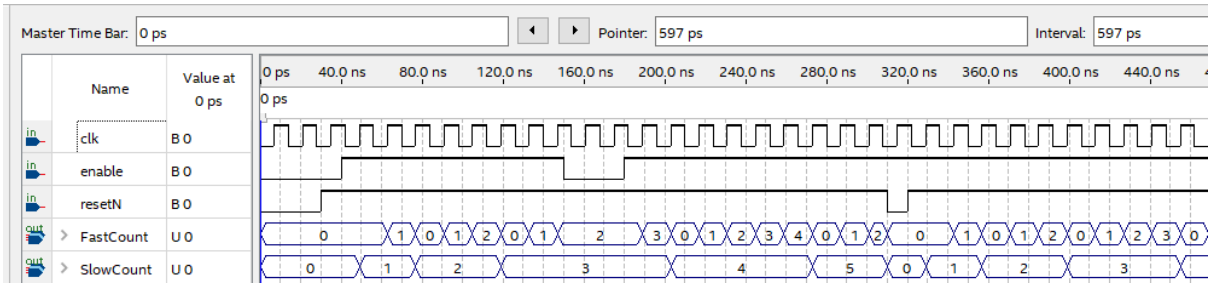
comparator code:

```
1  // Implements a simple equality one-bit out comparator
2  module comparator
3     (
4     // Input, Output Ports
5     input logic [3:0] vect1,
6     input logic [3:0] vect2,
7     output logic cmp
8     );
9  //-------------------------------------------------------
10 always_comb begin
11    if (vect1 == vect2)
12       cmp = 1'b1;
13    else
14       cmp = 1'b0;
15 end
16 //-------------------------------------------------------
17 endmodule
```

inflating counter code:

```
// Implements the inflating counter by instantiating
// two counters and a comparator

module inflating_counter
    (
    // Input, Output Ports
    input logic clk,
    input logic resetN,
    input logic enable,

    output logic [3:0] FastCount,
    output logic [3:0] SlowCount
    );

    logic enable_cnt; // internal variable - output of the comparator
//-------------------------------------------------------------------------------

    // Fast counter instantiation
    up_counter fastC(
             .clk(clk), .resetN(resetN), .enable(enable), .loadN(!enable_cnt), .enable_cnt(1'b1), .init(4'b0000), .count(FastCount)
            );

    // Slow counter instantiation
    up_counter slowC(.clk(clk), .resetN(resetN), .enable(enable), .loadN(1'b1), .enable_cnt(enable_cnt), .init(4'b0000), .count(SlowCount)
            );


    // Comparator instantiation
    comparator cmp
    (.vect1(FastCount), .vect2(SlowCount), .cmp(enable_cnt)
    );
//-------------------------------------------------------------------------------

endmodule
```

Inflating counter waveforms:



Hexadecimal to 7-segment conversion unit for FPGA LED, code:

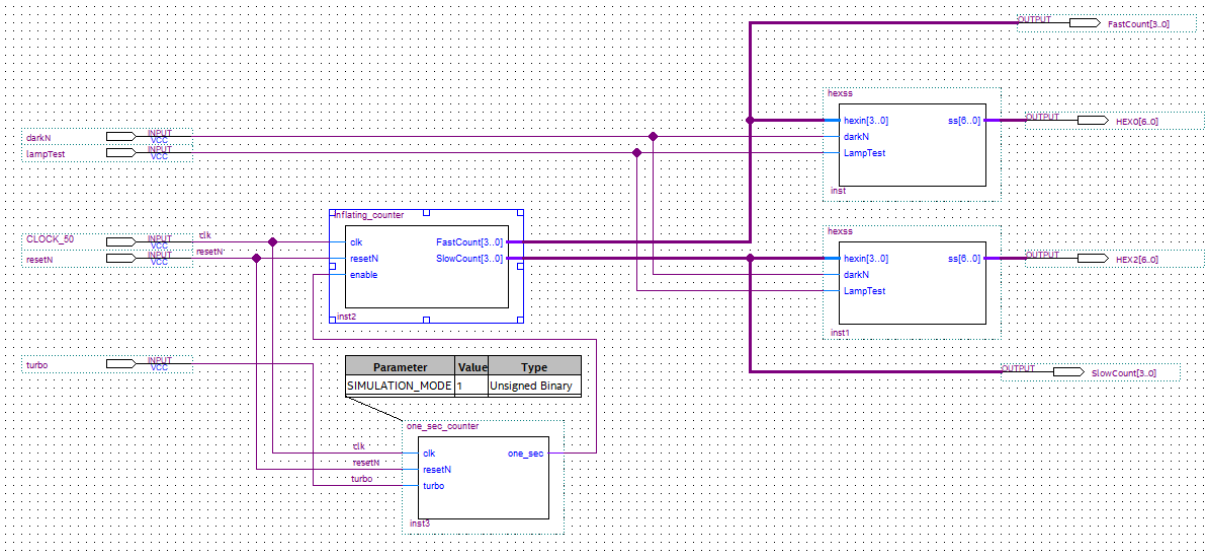
```
// Implements the hexadecimal to 7Segment conversion unit
// by using a two-dimensional array
module hexss
    (
    input logic [3:0] hexin, // Data input: hex numbers 0 to f
    input logic darkN,
    input logic LampTest,    // Aditional inputs
    output logic [6:0] ss    // Output for 7Seg display
    );

//-------------------------------------------------------------------------------------
// Declaration of two-dimensional array that holds the 7seg codes
    logic [0:15] [6:0] Seven_Seg = {7'b1000000, 7'b1111001, 7'b0100100,
    7'b0110000, 7'b0011001, 7'b0010010, 7'b1000010, 7'b1111000, 7'b0000000,
    7'b0010000, 7'b0001000, 7'b0000011, 7'b1000110, 7'b0100001, 7'b0000110,
    7'b0001110};

   always_comb begin
     if (darkN == 1 && LampTest == 0)
         ss = Seven_Seg[hexin];
     else if (darkN == 1 && LampTest == 1)
         ss = 7'b0000000;
     else if (darkN == 0)
         ss = 7'b1111111;
     else
         ss = 7'bxxxxxxx;
   end
//-------------------------------------------------------------------------------------
endmodule
```

one sec counter:

A frequency divider module, named one_sec_counter.sv, counts clock pulses and converts the very fast clock rate of the 10DE board (50 MHz) to a pulse at a slower rate of 1 Hz.

```
module one_sec_counter
   (
   // Input, Output Ports
   input  logic clk,
   input  logic resetN,
   input  logic turbo,
   output logic one_sec
   );

   int oneSecCount ;
   int sec ;       // gets either one seccond or Turbo top value

parameter logic SIMULATION_MODE  = 1 ; // 1 - simulation mode, 0 - real running mode

//       ------------------------------------------------  counter limit setting
   localparam oneSecVal_REAL = 32'd50_000_000; // for DE10 board
   localparam oneSecVal_SIM = 32'd20; // for quartus simulation
   localparam oneSecVal= SIMULATION_MODE ? oneSecVal_SIM : oneSecVal_REAL ; //select what to use
//       ------------------------------------------------

   assign  sec = turbo ? oneSecVal/10 : oneSecVal;  // it is legal to devide by 10, as it is done by the complier not by logic (actual transistors)


   always_ff @( posedge clk or negedge resetN )
   begin

      // asynchronous reset
      if ( !resetN ) begin
         one_sec <= 1'b0;
         oneSecCount <= 32'd0;
      end // if reset

      // executed once every clock
      else begin
         if (oneSecCount >= sec) begin
            one_sec <= 1'b1;
            oneSecCount <= 0;
         end
         else begin
            oneSecCount <= oneSecCount + 1;
            one_sec     <= 1'b0;
         end
      end // else clk

   end // always

endmodule
```

top bdf of inflating counter:



Final on FPGA: