ECE 551
Homework #2 Solution

1. **Saturation:**

Signal:	**Dir:**	**Description:**
unsigned_err[15:0]	in	16-bit **unsigned** error term to be reduced/saturated to 10-bits
unsigned_err_sat[9:0]	out	10-bit saturated version of unsigned_err[15:0]
signed_err[15:0]	in	16-bit **signed** number to be reduced/saturated to 10-bits
signed_err_sat[9:0]	out	10-bit saturated version of signed_err[15:0]
signed_D_diff[9:0]	in	10-bit **signed** version of a difference term used for derivative calculation
signed_D_diff_sat[6:0]	out	7-bit signed saturated version of signed_D_diff[9:0]

```
////////////////////////////////////////////////////////////////////////////////////
// For unsigned numbers if any of bits 15:10 are a one then the number
// is larger than we can represent in 10-bits so we just saturate to 3FF
////////////////////////////////////////////////////////////////////////////////////
assign unsigned_err_sat = (|unsigned_err[15:10]) ? 10'h3FF :    // sat to most +
                          unsigned_err[9:0];                    // otherwise take 9:0 as is

////////////////////////////////////////////////////////////////////////////////////
// For a signed number we need to look at the sign bit (MSB) and then look at
// bits 14:9.  If the incoming number is negative, and there is a zero in any of
// bits 14:9 then it is more negative than we can represent in 10-bits.  If it is
// positive and any of bits of 14:9 are set then it is more positive than we can
// represent in 10-bit.  Otherwise just use bits 9:0.
////////////////////////////////////////////////////////////////////////////////////
assign signed_err_sat = (signed_err[15] && ~&signed_err[14:9]) ?  10'h400 :      // most –
                        (~signed_err[15] && | signed_err[14:9]) ? 10'h3FF :      // most +
                        signed_err[9:0];                // OK as is

assign signed_D_diff_sat = (signed_D_diff[9] && ~&signed_D_diff[8:6]) ? 7'h40 :
                           (~signed_D_diff[9] && |signed_D_diff[8:6]) ? 7'h3F;
                           signed_D_diff[6:0] ;
```

2. **(30 pts)** Signed multiplication, and sign extension (Segway Math)

   This was done as an in class exercise, and we will be working on the math more in future exercises. This was your first cut at the math, and creating a testbench for it. For this testbench you had to pick some corner cases and compute what the result should be manually and compare. In a future exercise I will provide some random stimulus, and expected results in files and you will create a more thorough testbench.

```
wire signed [11:0] ptch_D_term;    // 7 + 5 = 12-bits needed
wire signed [9:0] ptch_P_term;     // ¾ takes same number of bits
wire signed [8:0] ptch_I_term;     // since we >>> it requires one less bit
wire signed [12:0] ptch_PID;       // should be 1-bit wider than widest term added
wire [11:0] ptch_PID_abs;  // when take ABS you need one less bit.

assign ptch_D_term = ptch_D_diff_sat * $signed(9);
assign ptch_P_term = (ptch_err_sat>>>1) + (ptch_err_sat>>2);
assign ptch_I_term = ptch_err_I>>>1;

assign ptch_PID = ptch_P_term + ptch_I_term + ptch_D_term;

assign ptch_PID_abs = (ptch_PID[12]) ? –ptch_PID : ptch_PID;
assign rev = ptch_PID[12];

assign mtr_duty = MIN_DUTY + ptch_PID_abs;
```

OF COURSE your solution should probably show some more commenting than mine. Do as I say…not as I do.

3. **(20 pts)** Using dataflow RTL implement a 4-bit wide adder that has a the following interface:

Signal:	Direction:	Description:
A[3:0]	in	Operand 1
B[3:0]	in	Operand 2
cin	in	Carry in
Sum[3:0]	out	Sum of operand 1 & 2
co	out	Carry out from addition of operands

Create a testbench that **exhaustively** tests all combinations of inputs. The testbench should also instantiate or implement a behavioral implementation of the adder to be used as the "golden reference". The testbench should be self checking and should error out and stop if there is a miscompare, or finish with a happy message if all goes well.

```
module add4bit(A,B,cin,Sum,co);

input [3:0] A,B;
input cin;
output [3:0] Sum;
output co;

assign {co,Sum} = A + B + cin;

endmodule
```

DUT code is simple, testbench follows:

The following testbench is one possible implementation. There are many ways one could have implemented the "golden check".

```
module add4bit_tb();

reg [9:0] stim;      // 1-bit wider than necessary so loop terminates

integer behave_sum; // golden result will be stored in integer
reg behave_co;

wire [3:0] Sum;      // connected to Sum output of DUT
wire co;             // connected to carry out of DUT

//////////////////////
// Instatiate DUT //
//////////////////////
add4bit iDUT(.A(stim[3:0]), .B(stim[7:4]), .cin(stim[8]),
             .Sum(Sum), .co(co));

////////////////////////////////////////////////
// Behavioral implementation of "golden" unit //
////////////////////////////////////////////////
always @(stim) begin
  behave_sum = rand_stim[3:0] + rand_stim[7:4] + rand_stim[8];
  behave_co = (behave_sum>15) ? 1'b1 : 1'b0;
end

initial
  for (stim=0; stim<512; stim=stim+1) begin     // for all possible input values
    #5;
    if ((behave_sum[3:0]!==Sum) || (behave_co!==co)) begin
      $display("ERR: DUT miscompare vs golden model\n");
      $stop();
    end
  end

endmodule
```

One of many possible implementations of self check for adder.

4.

a. Below is the implementation of a D-latch.

```
module latch(d,clk,q);
  input d, clk;
  output reg q;

  always @(clk)
    if (clk)
      q <= d;

endmodule
```

Problem with this implementation is the sensitivity list. I should also include **d**. If **d** changes during clk high it would not re-trigger as written.

Submit to the dropbox a single file called HW2_prob4.sv

D-FF with an active high synchronous reset.

```
always_ff @(posedge clk)
  if (rst)
    q <= 1'b0;
  else
   q <= d;
```

a. a D-FF with asynchronous active low reset and an active high enable.

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;
  else if (en)
    q <= d;
```

b. The file should contain the model of a SR FF with **active low asynchronous** reset. SR meaning it has a S input that will set the flop, and a R input that will reset the flop, and it maintains state if neither S or R are high. It also has active low async reset. This is a handy style flop that we will use frequently.

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;
  else if (R)
    q <= 1'b0;
  else if (S)
    q <= 1'b1;
```

c. I would like you to use the **always_ff** construct of System Verilog. The file should contain (as comments) the answer to this question: Does the use of the **always_ff** construct ensure the logic will infer a flop? Looking for a little more than a simple yes/no answer.

Use of **always_ff** does not ensure a flop is inferred. It simply means that if your logic does not infer a flop you will get a warning.

2) **(10 pts)** Find the datasheet for the LSM6DS3H. This is the inertial sensor we will use on the segway. In a text file called HW2_Prob5.txt answer the following questions:
3)
4) Does it use a SPI, I2C, or UART interface? SPI
5) Does the gyro output angular position, or angular rate? Angular rate (degrees/sec)
6) If we wanted an output data rate around 400 readings per second does it support that? Yes…there is a rate around 417Hz
7) How would we synchronize our Verilog with it…how would we know it has a new set of measurements ready for us? It has an interrupt pin that can be configured to interrupt when gyro or accel readings are ready.