ECE 551
Homework #2
Due: Weds Oct 3rd @ Class

Please Note: For this homework, you ***must*:**

- Use informative module/signal/port names whenever at all reasonable to do so.
- Work individually

1) **(20pts)** (***done as exercise in class Monday 24th***) Reducing and saturating results. With digital control schemes control input decisions are made based on error terms. Take our specific case of Segway control. If the Segway is leaning forward (pitched forward) more than desired we want to drive the motors hard in the forward direction. If this pitch error is small we want to make a slight adjustment (we want good granularity). If this pitch error is large we want to make a large adjustment. This implies we need a lot of bits to store the error, so we can have both fine granularity and large range. However, in control systems there gets to be a point where we don't care how large the error is we just need to know it is large and the system should do as much as it can to correct it. If the Segway is pitching forward way more than we want it to, we no longer care exactly how much, at that point we know we want to drive the motors forward really hard.

   So a wide error term could be reduced and saturated to a smaller number of bits to make down stream calculations narrower (fewer bits wide) and thus use fewer gates. This is something we will do in the controller math. OK…now that I have setup the motivation for the problem. Here is the problem:

   You are going to write a dataflow Verilog module with the following interface:

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]

   The name of the file should be ***saturate.v***

Write a test bench (***saturate_tb.v***) and instantiate the module. Apply several values to each input and observe the saturation of the outputs. You should choose several values that would cause the signed value to saturate in various ways, and also values that would not cause saturation.

Submit to the dropbox

a) saturate.v
b) saturate_tb.v
c) saturate_waves.png (or JPG) that shows you simulated.

2) (30 pts) (***started as exercise in class Friday 28th***) sign extension and signed multiplication. Sign extension is the exact opposite application as saturation (going from a number represented with a smaller number of bits to a number represented with a larger number of bits). We will make use of sign extension quite a bit in the balance controller math.

Verilog by default operates as unsigned. Signed multiplication is different than unsigned multiplication. To convince Verilog to make a signed multiplier both operands have to be of type signed, and the resulting signal the multiplication is stored in has to be of type signed.

If you are multiplying by a constant you can cast the constant to signed by using the **$signed()** method on the constant.

You are to create a Verilog module called ***duty.v*** using the following interface:

Signal:	**Dir:**	**Description:**
ptch_D_diff_sat[6:0]	in	7-bit signed number (represents derivative of error)
ptch_err_sat[9:0]	in	10-bit pitch error term (signed)
ptch_err_I[9:0]	in	10-bit integral of the error term (signed)
rev	out	Rev means we are driving motor in reverse (1=>reverse, 0=>forward)
mtr_duty[10:0]	out	Duty cycle of PWM drive to motor. MIN_DUTY + ABS(ptch_P_term + ptch_D_term + ptch_I_term). NOTE: this is a 11-bit quantity to handle any overflows of adding all these various terms together. Also note the absolute value. We drive our motor with a signed magnitude signal. So in this block we are essentially converting from 2's complement to sign/magnitude

The module will perform the following math:

ptch_D_term = ptch_D_diff_sat * 9

where **ptch_D_term** is internal signal. How wide does **ptch_D_term** need to be? How do you convince it to perform a signed multiplication?

ptch_P_term = (3/4) * ptch_err_sat

3/4 = 1/2 + 1/4 …Hmmm…can I get that result without using a multiplier?

ptch_I_term = ptch_err_I>>>1

Arithmetic right shift of **ptch_err_I**. Still a signed number.

ptch_PID = **ptch_P_term** + **ptch_I_term** + **ptch_D_term**

How wide should **ptch_PID** be? It is still a signed number.

The most significant bit of **ptch_PID** will form **rev**. If **ptch_PID** is negative we will be driving the motor backwards.

Finally **mtr_duty**[10:0] = **MIN_DUTY** + ABS(**ptch_PID**)

MIN_DUTY is an unsigned number and a parameter.

localparam MIN_DUTY = 11'h200

Create a testbench (***duty_tb.v***) and instantiate ***duty.v*** inside it. Choose stimulus to exercise and verify the design. You should choose both positive and negative numbers for **ptch_D_diff[5:0]** and **ptch_err_sat[9:0].**

Submit to the dropbox

a) duty.v
b) duty_tb.v
c) duty_waves.png (or JPG) that shows you simulated.

(**NOTE**: it is NOT intended that we will reuse any of this code directly in the project, but math similar to this will be done for the Segway control. You will see some of this again in a later HW in more detail.

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.

Submit to the dropbox

a. verilog for the DUT (**adder.v**) **(5pts)**
b. Turn in your verilog for the testbench (**adder_tb.v**)
c. Proof that you ran it to completion successfully.

4) **(20 pts)** (***started as exercise in class Monday Oct 1st)***

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
```

Is this code correct? If so why does it correctly infer and model a latch? If not, what is wrong with it?

Submit to the dropbox a single file called HW2_prob4.sv

a. The comments should answer the questions about the latch posed above.

b. The file should contain the model of a D-FF with an active high synchronous reset.

c. The file should contain the model of a D-FF with asynchronous active low reset and an active high enable.

d. 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.

e. 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.

5) **(10 pts)** Find the datasheet for the LSM6DS3H. This is the inertial sensor we will use on the quadcopter. In a text file called HW2_Prob5.txt answer the following questions:

a) Does it use a SPI, I2C, or UART interface?
b) Does the gyro output angular position, or angular rate?
c) If we wanted an output data rate around 400 readings per second does it support that?
d) How would we synchronize our Verilog with it…how would we know it has a new set of measurements ready for us?

Submit answers to these questions to the dropbox in a file called: HW2_prob5.txt